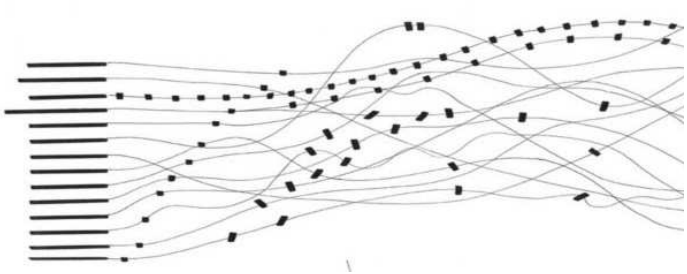
eleventh
international
conference
on
autonomous
agents and
multiagent
systems

AAMAS 2012

4th- 8th June 2012
Valencia

W21
Workshop on Spatial Computing (SCW)



Spatial Computing 2012

colocated with AAMAS

Valencia, 5 june 2012

Dr. Jacob Beal (BBN Technologies)
Dr. Stefan Dulman (Delft University)
Dr. Jean-Louis Giavitto (CNRS & IRCAM)
Prof. Antoine Spicher (LACL - Univ. Paris-Est Créteil)

Contents

Foreword

Many multiagent systems are *spatial computers* – collections of local computational devices distributed through a physical space, in which:

- the interaction between localized agents is strongly dependent on the distance between them, and
- the "functional goals" of the system are generally defined in terms of the system's spatial structure.

For example, spatial relationships are often used to organize the interactions between agents, at least in applications in which the problem and the space are intertwined. Furthermore, multiagent-based systems and their behaviors can be specified and analyzed relying on spatial notions like: location, region, frontier, neighborhood, obstruction, field, basin, communication, diffusion, propagation, etc.

Systems that can be viewed as spatial computers are abundant, both natural and man-made. For example, in wireless sensor networks and animal or robot swarms, inter-agent communication network topologies are determined by the distance between devices, while the agent collectives as a whole solve spatially-defined problems like "analyze and react to spatial temperature variance" or "surround and destroy an enemy."

On the other hand, not all spatially distributed systems are spatial computers. The Internet and peer-to-peer overlay networks may not in general best be considered as spatial computers, both because their communication graphs have little relation to the Euclidean geometry in which the participating devices are embedded, and because most applications for them are explicitly defined independent of the network structure. Spatial computers, in contrast, tend to have more structure, with specific constraints and capabilities that can be used in the design, analysis and optimization of algorithms.

The goal of **the 5th Spatial Computing Workshop** is to serve as an inclusive forum for the discussion of ongoing or completed work focusing on the theoretical and practical issues of *explicitly using space* in the design process of multiagent or multiactor systems.

Indeed, the handling of space often remains implicit and elementary in general, reflected also in its limited adoption in the myriad of domain-specific programming languages. We believe that progress towards identifying common principles, techniques, and research directions—and consolidating the substantial progress that is already being made—will

benefit all of the fields in which spatial computing takes place. And, as the impact of spatial computing is recognized in many areas, we hope to set up frameworks to ensure portability and cross-fertilization between solutions in the various domains.

The Spatial Computing Workshop provides a premier forum for sharing both research and engineering results, as well as potential challenges and prospects.

Workshop History

The domain of spatial computing is young but the last 6 years have shown a constant interest of researchers from various communities: parallelism, self-organization, complex systems, systems biology, swarm robotics, autonomic systems, amorphous computing, and at last but not least, multiagent systems.

The Spatial Computing Workshop (SCW) series has been established in 2008 after an initial Dagsthul seminar *Computing Media and Languages for Space-Oriented Computation*[1] in 2006 and a followup in Paris in 2008 *From Amorphous Computing to Spatial Computing* where the decision was made to begin the workshop. Since 2008, SCW has been a satellite workshop of the SASO conference[2]. This year, SCW is colocated with AAMAS. The past SCW have been held in:

- Venice[3] (2008)
- San Francisco[4] (2009)
- Budapest[5] (2010)
- Ann Arbor[6] (2011)

In 2009, a special issue of the ACM Transactions on Autonomous and Adaptive Systems (TAAS) were organized, where SCW participants and other researchers were invited to contribute.

In 2011, a special issue of The Computer Journal has been launched following the same open scheme.

[1] `http://www.dagstuhl.de/06361/`
[2] `http://www.saso-conference.org/`
[3] `http://projects.csail.mit.edu/scw08/`
[4] `http://radlab.cs.berkeley.edu/saso2009/workshops.html` and `http://scw09.spatial-computing.org/`
[5] `http://www.inf.u-szeged.hu/saso10/index.php?menu=workshop` and `http://scw10.spatial-computing.org/`
[6] `http://www.cscs.umich.edu/SASO2011/index.php?menu=workshop` and `http://scw11.spatial-computing.org/`

Current Trends in Spatial Computing

The papers presented at this workshop give a sampling of the current work in Spatial Computing. In the past editions of SCW, the following topics were discussed:

- Relationships between agent interaction and spatial organizations, self-organization, self-assemblies, collective motions;
- Characterization of spatial self-organization phenomena as algorithmic building blocks;
- Control theory approaches for designing dynamic spatial computing applications;
- Theoretical and practical limitations arising from spatial properties, understanding and characterization of spatial computing specific errors, analysis of tradeoffs between system parameters;
- Studies of the relationship between space and time - propagation of information through the spatial computer, and computational complexity;
- Languages for programming spatial computers and describing spatial tasks and space/time patterns;
- Methods for compiling global programs to local rules for specific platforms (so called global-to-local compilers);
- Suitable design methodologies and tools, such as novel domain-specific languages, for implementing, validating and evaluating spatial applications;
- Application of spatial computing principles to novel areas, or generalization of area-specific techniques;
- Device motion and control in spatial computing algorithms (e.g. relationship between robot speed and gradient accuracy in robotic swarms);
- Novel spatial applications, emphasizing parallel, mobile, pervasive, P2P, amorphous and autonomic systems;
- Testbeds and use-studies of spatial applications;
- ...

This year also, the papers selected for SCW reflect the diversity of spatial computing. They investigate a wide range of spatial programming features and applications, from distributed computing to ambient computing, from robotic and artificial life to design and music.

Without the help of the program committee, the workshop would not have come about and we would like to thank them for their involvement in SCW. Grateful acknowledgments are also due to the efficient logistic of the AAMAS organizers and especially Elizabeth Sklar, the AAMAS workshop chair, and to our supporting institutions: BBN Technologies, CNRS, Delft University, Inria MuSync team, Ircam and the RepMus team, UPMC, University of Paris-Est Créteil and the LACL lab.

We hope you will have as much fun and interest as us, in the reading of these proceedings.

April 2012

Dr. Jacob Beal (BBN Technologies)
Dr. Stefan Dulman (Delft University)
Dr. Jean-Louis Giavitto (CNRS & IRCAM)
Prof. Antoine Spicher (LACL - Univ. Paris-Est Créteil)

Program Committee

- Dr. Jonathan Bachrach (Other Lab, USA)
- Dr. Jacob Beal (BBN Technologies)
- Dr. Michel Banatre (Inria, France)
- Prof. Daniel Coore (University of West Indies, Mona)
- Prof. David De Roure (University of Southampton)
- Prof. Shlomi Dolev (Ben-Gurion University of the Negev)
- Dr. Rene Doursat (Institut des Systemes Complexes)
- Dr. Stefan Dulman (Delft University)
- Prof. Chris Dwyer (Duke)
- Prof. Amal El Fallah Seghrouchni (UPMC)
- Dr. Nazim Fates (INRIA)
- Dr. Jean-Louis Giavitto (CNRS & IRCAM)
- Prof. Erol Gelembe (Imperial College)
- Prof. Frederic Gruau (University Paris Sud)
- Prof. Guillaume Hutzler (University of Evry)
- Dr. Luidnel Maignan (INRIA Saclay, France)
- Prof. Mark Jelasity (Hungarian Academy of Sciences and Univ. of Szeged)
- Prof. Olivier Michel (LACL - Univ. Paris-Est, Créteil)
- Ulrik Pagh Schultz (University of Southern Denmark)
- Prof. Antoine Spicher (Univ. Paris 12)
- Prof. Christof Teuscher (Portland State University)
- Dr. Danny Weyns (K.U.Leuven, Belgium)
- Dr. Eiko Yoneki (University of Cambridge, UK)

Organizing Committee

- Dr. Jacob Beal (BBN Technologies)
- Dr. Stefan Dulman (Delft University)
- Dr. Jean-Louis Giavitto (CNRS & IRCAM)
- Prof. Antoine Spicher (LACL - Univ. Paris-Est, Créteil)

Schedule

09:00 Welcome

09:10-9:20 Introduction

09:20-09:50 *Lightweight Simulation Scripting with Proto*
Jacob Beal, Kyle Usbeck and Brian Krisler

09:50-10:20 *Arbitrary Nesting of Spatial Computations*
Antoine Spicher, Olivier Michel and Jean-Louis Giavitto

10:20-10:50 Morning coffee break

10:50-11:20 *Recursivity in Field-Based Programming: the Firing Squad Example*
Luidnel Maignan and jean-Baptiste Yunès

11:20-11:50 *Spatial computing for non-IT specialists* Steffan Karger, Agostino Di Figlia, Maurice Bos, Andrei Pruteanu and Stefan Dulman

11:50-12:20 *Spatial Structures Programming for Music*
Jean Bresson

12:20-13:00 Demonstrations

13:00-14:30 Lunch

14:30-15:00 *Decentralized spatial algorithm design*
Matt Duckham

15:00-15:30 *On the Space-time Situation of Pervasive Service Ecosystems*
Mirko Viroli and Graeme Stevenson

15:30-16:00 *Evolution of Controller-Free Molecular Motors from Spatial Constraints*
Jose-David Fernández, René Doursat and Francisco J. Vico

16:00-16:30 Afternoon coffee break

16:30-17:00 *Towards a Robust Spatial Computing Language for Modular Robots*
Ulrik Schultz

17:00-18:00 Demonstrations (continued) and Farewell

Lightweight Simulation Scripting with Proto

Jacob Beal
Raytheon BBN Technologies
Cambridge, MA, USA, 02138
Email: jakebeal@bbn.com

Kyle Usbeck
Raytheon BBN Technologies
Cambridge, MA, USA, 02138
Email: kusbeck@bbn.com

Brian Krisler
Raytheon BBN Technologies
Cambridge, MA, USA, 02138
Email: bkrisler@bbn.com

***Abstract*—Modern game engines make it easy to create complex realistic environments for entertainment or for training, but scripting the behavior of agents in these environments is still a major challenge. Spatial computing languages such as Proto [1] provide a possible solution, but need to be adapted for practical scripting use. We have begun to address this problem by linking Proto with the Unity game engine and by creating a Proto library for scripting the behavior of groups of agents. We validate our approach by demonstrating compact scripting of three complex agent interaction scenarios.**

I. Introduction

Modern game engines [2], provide the core components necessary to quickly produce simulations that just a few years ago required complex, custom solutions. The proliferation of these generic engines has led to the emergence of a new category of games, referred to as serious games [3]. The main focus area for serious games is training, where systems such as the US Navy VESSEL trainer [4] are used to reduce classroom lecture times and promote active learning.

Every game engine has a scripting environment that provides a language and core API for customizing interactions with and within the game. While these APIs are typically robust and allow for complete control of all objects within the game world, they are seriously limited in their support for quickly scripting behaviors for large groups of autonomous agents. For example, in the creation of a training game where a trainee would have to function in a large crowd, providing the movement flow and heterogeneous interactions typical of a realistic crowd would require many complex pieces of custom code, perhaps down to the level of individual agents. This requirement limits the inclusion of many autonomous agents in a training scenario.

In this paper, we address this problem by linking the Proto spatial computing language to Unity [5], a widely used modern game engine. We then create a library for scripting the behavior of groups of agents and demonstrate how our approach allows compact scripting for large groups of agents in a realistic simulation environment.

II. Background

Although there has been much previous work on agent behavior programming and simulation, there are significant gaps in the capabilities of existing solutions. These current solutions can be classified into three categories: single-agent behavioral models, multi-agent toolkits, and spatial computing platforms. A detailed review discussing many of the approaches described in this section can be found in [6].

Many game engines simply use conventional programming languages (or their own domain-specific variants) for their scripting languages. For example, Unity uses scripts written in JavaScript, C#, and Boo. More sophisticated single-agent behavioral models include conceptual models of agent behavior and agent frameworks (for implementing agent behaviors). Conceptual behavior models, such as the Belief-Desire-Intent (BDI) agent model [7], offer high-level descriptions of agent internals. Agent frameworks (often called "agent architectures") described thoroughly in [8] and [9], provide tools (e.g., agent administration, messaging, mobility, logging, etc.) for implementing agent behaviors. These frameworks and behavioral models, however, rarely provide aggregate programming/modeling tools that are useful for MAS control.

Multi-agent System (MAS) modeling and simulation toolkits tend to focus on interactions: both inter-agent interactions and interactions between agents and their environment. MAS modeling and simulation toolkits include languages for modeling the MAS, and tools for simulating the running agent system. For example, NetLogo [10] extends the Logo language to allow agent coordination and provides a graphical tool for simulating the agent behaviors. Most existing MAS modeling and simulation toolkits lack realism in their simulation environments, and therefore do not provide language features for realistic behavior. Furthermore, most MAS toolkits lack features for spatial aggregate programming, which we describe next, which enable scalable descriptions of aggregate behavior (i.e., the number of agents don't need to be specified *a priori*).

A more-recent approach is *spatial computing*, which assumes communication is constrained to agents near one another in space. The implication of this assumption is that it becomes necessary to consider the spatial structure of the system in planning the solution. Proto [1] is a purely-functional LISP-like language that is designed with spatial constructs (i.e., operations to measure and manipulate space-time, compute spatial patterns, and evolve dynamically). General purpose spatial languages such as Proto or MGS [11] are capable of elegantly and concisely describing aggregate MAS behavior (sometimes labeled "emergent behavior") [12], but often have unusual programming models. For example, Proto is a purely

Work partially sponsored by DARPA; the views and conclusions contained in this document are those of the authors and not DARPA or the U.S. Government.

functional language and does not offer the imperative-style MAS scripting that is familiar to game-based agent behavior developers. Furthermore, the simulators used for running and testing spatial languages tend to lack the realism that is available from recent physics simulators and game engines.

III. Approach

Our approach for creating a scalable aggregate scripting language for realistic simulation environment has three components: *(1)* the Unity game engine [5] provides realism in the simulation environment (e.g., terrain modeling, realistic physics simulation, entity modeling), *(2)* the Proto spatial computing language provides constructs for scalable aggregate programming, and *(3)* agent behavior scripting is facilitated by a novel Proto library comprising group behavior primitives and a novel macro library for imperative-style scripting.

A. Connecting Proto and Unity

Proto has three main components: (1) the spatial language, (2) a global-to-local compiler which accepts a global behavior description (in Proto language) and outputs a virtual machine (VM) binary for the Proto VM, and (3) the VM that interprets Proto VM instructions on each device. In order to be able to execute global Proto programs from within the realistic Unity simulation environment, we first make the Proto compiler invokable from within Unity. Next, we create an implementation of the Proto VM for reading information from and performing operations upon Unity agents. Finally, we create an interface for developers to control parameters of the Proto-Unity plugin.

1) Invoking the Proto Compiler: Proto uses a global-to-local compiler to convert global behavior descriptions into local (i.e., per-device) programs. It is important that this compiler be integrated into the final solution so that end-users can write programs for groups of agents within the modeling and simulation toolkit.

The reference implementation of the Proto compiler is written in C++ and Unity has a C/C++ API for its plugins, so one option would be to directly integrate the Proto compiler into a Unity plugin. This would have required maintaining a branch of the Proto compiler with a Unity-friendly interface, however. Instead, we created a simpler plugin that invokes an external installation of the standard Proto compiler. Thus, Unity, supplied with a Proto program, invokes the external Proto compiler on that program and receives in return the Proto VM instructions (a.k.a., Proto opcodes) that specify how each agent should act.

2) A Proto VM Implementation for Unity: Next, we need a mechanism for controlling agents within Unity according to the behaviors described by the local Proto VM instructions. The Proto reference implementation already contains a VM suitable for most environments—requiring the developer to implement only a small set of platform-specific functions (e.g., how the machine allocates memory, broadcasts messages to other devices, etc.). Likewise, the continuous time model of Proto programs means there is no problem matching simulation rates: the VM execution rate can simply be derived from its Unity environment. Importing the Proto VM to Unity was not significantly different or more difficult than prior imports on various embedded platforms: to do so, we constructed a Unity plugin that implements the required platform-specific functions using tools from the Unity API. For example, one such function uses the Unity utility for computing the distance between Unity agents to implement a unit-disc communication model. Of course, this model could be extended to incorporate other information available from Unity (e.g., line-of-sight) for improved realism.

3) An Interface to the Proto-Unity Plugin: Finally, we created an interface for controlling the Proto-Unity plugin. This engineering interface is not meant for end-users, but instead is designed to help agent script developers by providing functionality similar to that of the reference implementation of the Proto simulator. For example, the interface can show the network topology of the Unity agents by drawing lines between the agents within communication range and allows the developer to change the devices' communication radii on-the-fly. This allows a developer to "tweak" simulation parameters during development, then set them and remove the interface when the simulation is finished and provided to users.

B. Group Behavior Primitives

We next need a library of "primitives" for group behaviors—simple ways of describing what we want a collection of agents to do. These will be the basis for the agent scripts that we build. We build this library after the fashion of [13], as Proto functions that compute vector fields for the desired motion of agents. We can then produce complex behaviors by mixing these vector fields together in various ways.

We have created an initial library of eight behaviors. Figure 1 shows examples of these behaviors being applied to agents in Unity using the Proto/Unity bridge. Here we present only the API for the behaviors; their implementation is similar (or in some cases identical) to code presented in [13].

(`random-walk`)		
Parameter	*Type*	*Description*
RETURN	`TUPLE`	Vector direction for agent to move

Random-walk moves each agent in a random direction. This is like `brownian` in [13], except that speed is also randomized.

(`flock` `DIRECTION`)		
Parameter	*Type*	*Description*
`DIRECTION`	`TUPLE`	Preferred direction to flock toward
RETURN	`TUPLE`	Vector direction for agent to move

The `flock` behavior, also from [13], moves agent groups by repelling the closest agents, aligning with moderately-proximate agents, and weakly attracting distant agents. This allows a group of agents to "flock" together in partially-coherent group motion. The `DIRECTION` argument guides the motion of the flock with a preferred direction supplied to some or all members, as investigated in [14].

(`flock-to` `LOCATION`)		
Parameter	*Type*	*Description*
`LOCATION`	`TUPLE`	Coordinates to flock to
RETURN	`TUPLE`	Vector direction for agent to move

(a) Random-Walk (b) Flock/Flock-To (c) Disperse/Scatter (d) Toward (e) Cluster-By

Fig. 1. Examples of agents being controlled by group behavior primitives written in Proto.

The `flock-to` behavior is like `flock`, except the agents move coherently to a location, rather than toward a direction.

(disperse)		
Parameter	*Type*	*Description*
RETURN	TUPLE	Vector direction for agent to move

The `disperse` behavior, our last adaptation from [13], repels agents away from one another with a force proportional to the inverse square of the distance separating them.

(scatter DIRECTION)		
Parameter	*Type*	*Description*
DIRECTION	TUPLE	Vector biasing scatter direction
RETURN	TUPLE	Vector direction for agent to move

The `scatter` behavior is much like `disperse`, except that agents do not slow down when they start getting far apart and they have a directional bias, `DIRECTION`.

(toward TARGET)		
Parameter	*Type*	*Description*
TARGET	BOOLEAN	Boolean indicator that is `true` if an agent is a target
RETURN	TUPLE	Vector direction for agent to move

The `toward` behavior finds the direction toward the mean location of all neighbors with a `TARGET` property.

(away-from TARGET)		
Parameter	*Type*	*Description*
TARGET	BOOLEAN	Boolean indicator that is `true` if an agent is a target
RETURN	TUPLE	Vector direction for agent to move

The `away-from` behavior is the inverse of `toward`.

(cluster-by GROUP-ID)		
Parameter	*Type*	*Description*
GROUP-ID	INTEGER	Identifier for an agent group
RETURN	TUPLE	Vector direction for agent to move

Finally, `cluster-by`, sorts agents into groups: all agents repel each other weakly and are strongly attracted to others with the same `GROUP-ID` identifier. This will tend to separate the group into clusters by identifier, though if the agents are widely scattered, there may be more than one cluster for any given identifier.

C. Agent Scripting Library

The last ingredient needed is a means of composing together these group behavior primitives to form useful agent behavior scripts, which will typically be much more complicated. Proto's native approach is one of purely functional composition—mathematically elegant, but not well suited for the way that simulation designers often like to think. Instead, we would like to be able to talk about a script in terms of concepts like particular groups being assigned particular behaviors, responding to triggers, or progressing through a planned sequence one stage at a time.

Technically, Proto's functional model can already provide all of these capabilities. The problem is that the code to do so is often awkward and does not "look" like the kind of state-based programming that is more familiar for this sort of scripting. Fortunately, Proto has recently been extended with a capability for syntactic macros. We use this macro programming facility to create new syntactic constructs suitable for agent scripting. The macros transform these new syntactic constructs into implementation in terms of standard Proto primitives.

Our initial agent scripting library comprises five constructs, selected as examples of group and individual behavior selection and sequencing; other important categories not yet included are behavior planning, collective decision making, etc. In our initial library, `group-case` and `where` assign behaviors to groups of agents, `on-trigger` sets up a triggered action for a group of agents, `priority-list` assigns behavior based on the relative importance of competing priorities, and `sequence` moves a group of agents through a planned sequence of actions. All of these are defined with the assumption that the return value is intended to be a vector field specifying the movement of agents. We will now detail each of these constructs in turn, then demonstrate their use with the examples in the next section.

The syntax of the `group-case` construct is:

```
(group-case
 (behavior-of MEMBERSHIP-TEST BEHAVIOR
 (behavior-of MEMBERSHIP-TEST BEHAVIOR
  ...
 (default BEHAVIOR)...)))
```

This operates much like an ordinary case statement: each `MEMBERSHIP-TEST` must be a boolean-value expression, and agents use the `BEHAVIOR` of the first `behavior-of` case they match. If an agent is not a member of any group, then it uses the `default` group's `BEHAVIOR`.

Within each `behavior-of` construct, there is a special variable `in-group` defined. Computations for a group's behavior normally extend over both agents in the group and agents outside of the group, allowing agents to react to information from others outside of their group. The `in-group` variable is true only for those agents in the group, and can thus be used to restrict computation to only within the group.

The `where` construct is a good way to do such a restriction:

```
(where TEST BEHAVIOR)
```

This computes BEHAVIOR over the set of agents where TEST is true, much like the standard Proto if construct, except that all other devices default to a tuple of zeros.

The on-trigger construct has identical syntactic structure:

```
(on-trigger TRIGGER BEHAVIOR)
```

Its function, however, is to enable a BEHAVIOR in a dormant group of agents as soon as TRIGGER becomes true for at least one member of the group. Once enabled, the group stays enabled and continues to act.

The syntax of the priority-list construct is:

```
(priority-list
 (priority NAME TEST BEHAVIOR
 (priority NAME TEST BEHAVIOR
 ...)))
```

Each agent walks the list of priorities in descending order, treating the first entry as highest priority. When a TEST evalutes to true, the agent executes the associated BEHAVIOR. If no priority holds, then the agent does nothing.

Finally, the sequence construct, which moves agents through a sequence of actions over time, is:

```
(sequence
 ([stage|group-stage] NAME ACTION TERMINATION
 ([stage|group-stage] NAME ACTION TERMINATION
 ...
 [end-sequence|repeat])...))
```

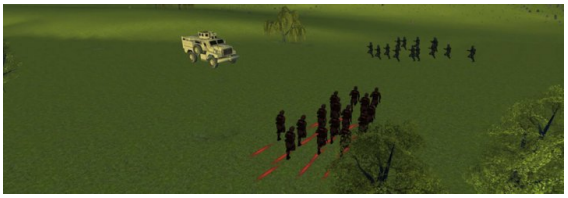
Agents transition individually through stage constructs and transition collectively out of group-stage constructs. The sequence begins with the first stage, executing ACTION until the TERMINATION condition is met. When an agent finishes a stage, it just moves on and begins executing the next stage's ACTION. For a group-stage, on the other hand, the agent also informs all neighboring agents, which move on to the next stage and inform their neighbors as well, and so on until all agents with reach of communication have changed stages. Thus, a group-stage terminates when *any* agent in the group reaches its TERMINATION condition.

When agents reach the final action in the sequence, their behavior depends on the final keyword. If the keyword is end-sequence, the agents stop moving; if it is repeat, the sequence begins again. Optionally, the keyword ongoing may be substituted for the last stage's TERMINATION, in which case the last stage continues indefinitely instead.

Although these five constructs are just a beginning of the type of constructs that are necessary to make up a full-fledged agent scripting library, they demonstrate that Proto macros can allow more "natural" scripting for agent behaviors.

IV. Validation

We now have an agent scripting library written in Proto and the ability to execute Proto programs in Unity—all of the ingredients necessary for validating our approach to lightweight simulation scripting. In this section, we demonstrate the power of our approach by constructing three simulations where groups of agents need to interact and to coordinate their behaviors with one another.



(a) Red team advances on Blue team

(b) Blue team notices incoming Red team

(c) Blue team scatters

Fig. 2. The red-advance script running on 30 agents.

For these simulations, we consider environments with two teams of agents: "Red team" aggressors and "Blue team" defenders. In each simulation, agents from both teams are placed onto a geo-typical terrain where they can then execute their group behaviors within the physics and terrain based constraints of the environment. We run these simulations with 10 to 30 agents; since the code is written in Proto, however, the same simulations can be executed on any number of agents.

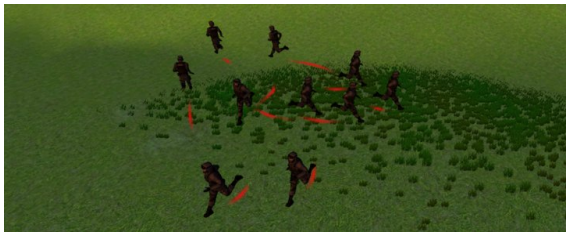
A. Red Advances on Blue

We begin with a simple scenario where Red team advances on Blue team and Blue scatters and flees when Red gets close:

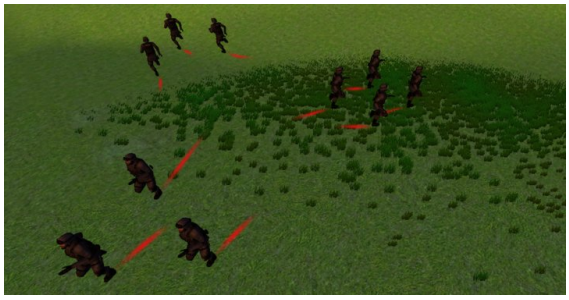
```
(def red-advance (red-team blue-team)
  (group-case
   (behavior-of red-team       ;; Red team behavior:
    (where in-group
     (flock-to (tup 0 0)))     ;; go to Blue starting location
   (behavior-of blue-team      ;; Blue team behavior:
    (on-trigger (can-see red-team) ;; when Red is near...
     (scatter (away-from red-team))) ;; flee from Red!
   (default (tup 0 0))))))
```

We do this by using the group-case construct to specify behavior by team. For Red team, we use flock-to to advance coherently towards the starting location of Blue team. For Blue team, we use on-trigger to scatter when any Blue agent notices an advancing Red, using the bias argument to making sure that the Blue agents move away-from Red. If any agent is not on either Red or Blue team, it does nothing. Figure 2 shows agents executing this scenario in Unity.

(a) Team moving as a group



(b) Team breaking up into three sub-groups



(c) Sub-groups moving to different destinations

Fig. 3. The `deploy` script running on 10 agents.

B. Red Deploys from a Vehicle

The next scenario has Red team deploying out of an armored transport vehicle into three squads:

```
(def deploy (squadID)
  (sequence
   (stage leave-vehicle          ;; First stage:
    (flock (tup -1 0 0))           ;; move left...
    (timeout 20)                   ;; ... for twenty seconds.
   (stage group-by-squad         ;; Second stage:
    (cluster-by squadID)           ;; group into squads...
    (timeout 50)                   ;; ... for fifty seconds.
   (stage deploy-to-destination ;; Third stage:
    (group-case        ;; Each squad goes to a different location:
     (behavior-of (= squadID 0)    ;; First squad ...
      (flock-to (tup 50 100))      ;; ... goes to (50, 100)
     (behavior-of (= squadID 1)    ;; Second squad ...
      (flock-to (tup -200 0))      ;; ... goes to (-200, 0)
     (behavior-of (= squadID 2)    ;; Third squad ...
      (flock-to (tup -100 -100))   ;; ... goes to (-100, -100)
     (default (tup 0 0))))))
    ongoing                  ;; Sequence doesn't end or repeat
   end-sequence)))))
```

Here, we use the `sequence` construct to break the deployment into three phases. First, the agents all `flock` for 20 seconds to leave the vehicle together. Next, the agents use `cluster-by` to sort themselves out into squads, giving 50 seconds for the squads to organize themselves. Finally, we use `group-case` to have each of the three squads `flock-to` its own destination. Figure 3 shows agents executing this scenario in Unity.

(a) Blue team on patrol — looking for Red team.

(b) Blue team members break off to chase Red team.

Fig. 4. The `patrol-encounter` script running on 30 agents.

C. Red Tries to Sneak Past a Blue Patrol

Our third scenario is the most complex: Blue team is trying to defend against Red team while patrolling a regular pattern. Meanwhile, Red team is trying to pass through the area that Blue team is guarding without being caught.

We first define the patrol pattern to be used by Blue team:

```
(def patrol ()
  (sequence
   (group-stage checkpoint-1 ;; First stage:
    (flock-to (tup 100 50))       ;; Go toward (100, 50) ...
    ;; ... until somebody in the patrol is within 5 meters of the place ...
    (< (vlen (- (coord) (tup 100 50))) 5)
   (group-stage checkpoint-2 ;; Second stage:
    (flock-to (tup 0 50))         ;; ... now go to (0, 50) ...
    (< (vlen (- (coord) (tup 0 50))) 5)
   (group-stage checkpoint-3 ;; Third stage:
    (flock-to (tup 50 -50))       ;; ... and then to (50, -50) ...
    (< (vlen (- (coord) (tup 50 -50))) 5)
   repeat)))))                    ;; ... and repeat
```

Here, we use a repeating `sequence` construct to define a cyclic patrol around three checkpoints. For each checkpoint, the agents use `flock-to` to move as a group to that checkpoint. Once any agent in the group reaches the checkpoint, the `group-stage` construct means its information will spread, causing the whole group to head for the next checkpoint even if some members have not yet reached the current checkpoint.

We then use this script as a behavior in the scripts for the overall encounter between Red team and Blue team:

```
(def patrol-encounter (red-team blue-team)
  (group-case
   (behavior-of red-team ;; Red team behaviors
    (priority-list
     (priority defend-self
      (can-see blue-team)              ;; if Blue shows up...
      (scatter (away-from blue-team));; ... then run away
     (priority invade                  ;; otherwise,
      (timeout 500)                ;; when the script says start
      (flock-to (tup 200 0))))) ;; try to pass by Blue team
```

```
(behavior-of blue-team ;; Blue team behaviors
 (priority-list
  (priority attack-red
   (can-see red-team)            ;; if Red shows up...
   (let ((dir (toward red-team))) ;; ... then track ...
     (where in-group (flock dir)));; ... and chase them
  (default                       ;; otherwise,
   (where in-group (patrol)))));; walk your patrol route.

(default (tup 0 0))))))
```

As before, we use `group-case` to specify a behavior for each team. We now also use `priority-list` to give each team multiple possible behaviors, depending on circumstances. Red team begins to invade the area 500 seconds after the simulation starts, attempting to `flock-to` its target. If a Red team agent encounters Blue team, however, this goal will be pre-empted by the goal of defending itself, and it will `scatter`, fleeing `away-from` Blue team.

The Blue team has a complementary `priority-list` script: when there are no Red team agents nearby, they default to patrolling on the three-checkpoint pattern that we defined above. If a Red team agent is nearby, though, they will break off to attack, using `flock` to move toward nearby red-team agents. Figure 4 shows agents executing this scenario in Unity.

Taken together, these demonstrate the power of our approach to scripting of agent-based simulations. Unity provides realistic physics simulation, while Proto and our new agent scripting library allow for compact scripting of scenarios in which groups of agents engage in many types of interactions.

The scenarios presented are remarkably compact in code, requiring only 8 lines, 19 lines, and 31 lines respectively. The same scenarios written in a conventional scripting language would typically take at least an order of magnitude more code. We can measure this in some cases by comparing against similar scripts available on the Unity community site, `http://www.unifycommunity.com/`. For example, the Proto `flock` code presented above takes only 12 lines, while a Unity JavaScript version takes 77 lines, yet must "cheat" in its calculations and can only run a single flock. Similarly, a single-agent waypoint following Program in Unity is implemented with 65 lines of JavaScript code, while it takes only 7 lines of Proto code to script coherent waypoint following for groups of agents. Even single agent functions are often much simpler in Proto: a Unity C# script for a wandering agent requires 40 lines, while the equivalent Proto function `dither` (part of the standard Proto library) requires only 5 lines.

Also of note is the relatively light computational burden of Proto; in the scenarios presented, the limiting factor on Unity simulation speed appears to be the cost of rendering the agents, with the cost of Proto computation and communication insignificant.

While not yet a definitive study, these results do clearly indicate that it is reasonable to expect large benefits from scripting agent-based simulations in Proto. We believe such a drastic reduction in size is likely to be due to two factors. First, as a functional programming language, Proto tends to produce more compact code. More importantly, however, Proto's ability to program aggregates and to execute routines on subgroups makes it just as easy to script a group behavior as a behavior for a single individual. The relative importance of these two factors, however, is not yet established.

V. Contributions

We have presented a novel approach to construction of agent-based simulation, based on the integration of the Unity simulation engine with the Proto spatial computing programming language. We have developed a library of agent group behaviors and scripting constructs aimed at programming this environment, and have demonstrated that the combination allows succinct specification of complex simulation scenarios with large numbers of interacting agents.

While the results presented in this paper demonstrate the potential for major improvements in agent-based simulation programming, there is much more that can be accomplished. Future work includes better integration between Proto and Unity, refinement and extension of the behavior library and scripting constructs, and construction of virtual sensors to allow Proto-controlled agents access to more information available from the Unity simulator, such as terrain properties, physical contact and line-of-sight.

References

[1] J. Beal and J. Bachrach, "Infrastructure for engineered emergence on sensor/actuator networks," *IEEE Intelligent Systems*, 2006.

[2] M. Lewis, J. Jacobson, and C. based Games, "Game engines in scientific research," 2002.

[3] T. Susi, M. Johannesson, and P. Backlund, "Serious games – an overview," 2007.

[4] T. Hussain, B. Roberts, C. Bowers, J. Cannon-Bowers, E. Menaker, S. Coleman, C. Murphy, K. Pounds, A. Koenig, R. Wainess, and J. Lee, "Designing and developing effective training games for the US Navy," in *2009 Interservice/Industry Training, Simulation and Education Conference.*, 2009.

[5] "Unity — 3D game engine," Available: `http://unity3d.com/`, Retrieved March 4, 2012.

[6] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," *CoRR*, vol. abs/1202.5509, 2012.

[7] A. Rao and M. Georgeff, "BDI agents: From theory to practice," in *Proceedings of the first international conference on multi-agent systems (ICMAS-95)*. San Francisco, 1995, pp. 312–319.

[8] W. C. Regli, I. Mayk, C. J. Dugan, J. B. Kopena, R. N. Lass, P. J. Modi, W. M. Mongan, J. K. Salvage, and E. A. Sultanik, "Development and specification of a reference model for agent-based systems," *Trans. Sys. Man Cyber Part C*, vol. 39, pp. 572–596, September 2009. [Online]. Available: http://portal.acm.org/citation.cfm?id=1656816.1656823

[9] D. N. Nguyen, K. Usbeck, W. M. Mongan, C. T. Cannon, R. N. Lass, J. Salvage, and W. C. Regli, "A methodology for developing an agent systems reference architecture," in *11th International Workshop on Agent-oriented Software Engineering*, Toronto, ON, May 2010.

[10] E. Sklar, "Netlogo, a multi-agent simulation environment," *Artificial life*, vol. 13, no. 3, pp. 303–311, 2007.

[11] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," Univerite d'Evry, LaMI, Tech. Rep. 103-2004, 2004.

[12] K. Usbeck and J. Beal, "An agent framework for agent societies," *Systems, Programming, Languages and Applications: Software for Humanity*, 2011.

[13] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous space programs for robotic swarms," *Neural Computing and Applications*, vol. 19, no. 6, pp. 825–847, 2010.

[14] I. Couzin, J. Krause, N. Franks, and S. Levin, "Effective leadership and decision-making in animal groups on the move," *Nature*, vol. 433, no. 7025, pp. 513–516, 2005.

Spatial Structures Programming for Music

Jean Bresson
UMR STMS: Ircam-CNRS-UPMC
Paris, France
Email: jean.bresson@ircam.fr

***Abstract*—We survey works and applications of visual programming for the control of sound spatialization in the OpenMusic computer-aided composition environment. Several salient aspects of the control and spatialization processes are described, such as the high-level description of spatial sound scenes, their unfolding in musical or computational time flows, as well as the concept of spatial sound synthesis.**

I. Introduction

Considering space in music is an long-lasting concern which has been particularly put forward during the last 50 years with the development of computer and audio technologies [19]. After pioneering works on early analogue systems by composers such as K. Stockhausen or E. Varèse, the interest for composers to integrate sound spatialization in their work significantly increased as digital technologies for spatial audio progressed and made it more accessible [27], [30]. With this technology, instrumental music performances can be enhanced by new techniques for sound diffusion in concert halls, and sounds in electro-acoustic works can be spatially composed in real or virtual rooms.

With the term *spatialization* we refer to the localisation and movements of sound sources, but also to their orientation or directivity patterns, to the acoustic properties of a room, or to any other more or less abstract elements related to the diffusion of sound sources and the inclusion of space as a musical parameter in a compositional context [5]. Examples of spatial audio technologies available today range from multi-speakers amplitude panning [33], which can now be controlled at high rate and resolution for large numbers of sound sources and channels, to more advanced systems and hardware equipments such as higher-order ambisonics [14] or wave-field synthesis [6].

Our interest here, however, is not exactly in spatial audio but in its control and integration in compositional frameworks.

Digital audio workstation plug-ins for sound spatialization are commonplace today, and are used to control the virtual localisation of sounds during sound mixing or montages. Powerful spatialization tools are available as well for music and sound processing environments like Max/MSP [21], [34]. Interestingly, recent efforts are also being done to provide high-level interfaces independent from the low-level rendering technologies [18], thereby permitting users to focus on control (and further, compositional) issues. However, systems for creating (composing) spatial structures, possibly used in a subsequent phase to parametrize the aforementioned control tools, are fewer (see for instance [23], [31], [41]).

The works we present take place in the OpenMusic visual programming environment and in the general context of computer-aided music composition. After a quick presentation of this background, we will try to underline the interest and possibilities offered by the integration of sound spatialization in high-level compositional frameworks, and describe some tools and concepts developed recently for this purpose.

II. OpenMusic, Visual Programming and Computer-Aided Music Composition

Visual programming is a relatively widespread approach to computer-aided composition in contemporary music [3]. OpenMusic (OM) is a visual programming language designed for music and used by composers, musicologists or computer music researchers to create or transform musical material [4], [11]. Programs in OM (also called *patches*) are represented and manipulated as directed acyclic graphs made of boxes and connections. A patch represents a functional expression which can be constructed and evaluated on-demand in order to produce or transform musical material (Figures 1 and 3 are examples of OM patches). OM is implemented in Common Lisp and OM patches have therefore a direct correspondence to Lisp expressions. Conversely, any Lisp code or process can be integrated in an OM patch in a straightforward way. The main programming paradigm in OM is therefore functional (most of the functional constructs available in Lisp can be implemented in visual programs, such as higher-order, recursive function, etc. [10]) although object-oriented features are also available in the visual language.

OM also contains a number of domain-specific data structures and associated graphical editors, allowing to visualize and manipulate the data involved in the visual programs and thereby providing an important input and feedback between the user/programmer and the program being created. It may be important, however, to differentiate functional environments such as OM from real-time (highly interactive) environments more commonly used in music (e.g. [32]). In OM in principle, no temporal control or external clock influences the patch execution, which is a static (declarative), and globally "out-of-time" description of a musical structure, generally including time as an autonomous and unconstrained dimension.

Advanced formalisms and compositional processes can therefore be carried out in OM in order to generate material or experiment with computation and computer modelling in diverse musical contexts [1], [9].

III. Spatial Concepts in Computer-Aided Composition

There exist several and varied ways in which spatial attributes can be linked to musical aspects of a work in a musical programming environment like OpenMusic. A first example, which actually does not deal with spatialization, is for instance given in [20] where the author uses spatial relations coming from the transcription and processing of architectural sketches and maps to produce musical parameters.

Conversely, and more related to our present concern, any kind of musical or extra-musical data can be processed in OpenMusic and generate attributes or parameters for a spatialization process. This approach has been experimented in the late nineties for instance to control the MusicSpace constraint-based spatialization system [16] with the *OpenSpace* project [17], or the Ircam *Spatialisateur* [21] with *OMSpat* [26]. The main interest here is that spatialization parameters (mostly, the position and movements of the sound sources in space) can be precisely controlled and set in accordance and relation to the other parameters of a compositional process, and therefore of the created musical material. A pioneering work in this respect, carried out in OpenMusic, was B. Ferneyhough's piece *Stellæ for failed times* [25].

Usually two different approaches can be observed in the control of sound spatialization. The first one focuses on the sound sources distribution among the different channels or speakers. This approach is adapted (and generally specific) to particular rooms and speaker set-ups: the composer "thinks" in terms of these speakers and devises how sound sources are allocated, and possibly move among them [40]. A second approach rather focuses on perception, that is, on positions where sound sources shall be located, and spatialization processors work at rendering this perceptual idea in the actual room and with a given speakers set-up. In this case, composers think in term of absolute positions and trajectories. In principle (but rarely in fact) this approach in the compositional domain can be independent of the room and set-up. The recent works carried out in OM, presented in the following sections, mostly focus on the latter approach, describing spatial sound scenes and processes in terms of positions and trajectories (although they do not invalidate the former one).[1] This idea has also been extended to the micro-level of sounds with the concept of *spatial sound synthesis* described in Section VIII.

IV. Spatial Scene Description in Visual Programs

Spatial sound scenes[2] are represented in OM visual programs by matrices, where one dimension stands for the different sources, and the orthogonal dimension represents the parameters used for a given spatialization process (these parameters may change depending on the application and spatialization technique, although they often include at least 2D or 3D position information). These matrices are included and instantiated in functional programs as described in section II (see Figure 1–a). The generative processes can be arbitrarily complex and involve varied formalisms and programming concepts (iterative processing, randomization, higher-order, etc.—see Figure 1–b).

V. Time and Trajectories

Time is a fundamental parameter in music and therefore needs a specific consideration in spatialization processes as well. Every parameter is subject to possible changes or continuous evolution. In particular, positions of the sound sources are often rather considered in terms of such evolution: in this case, spatial coordinates will not be represented by values but with sampled curves (determined in the generative processes either point-wise or as functional specification), or contained at a higher level as "trajectory" objects.

The *trajectory* objects in OM aim at providing self-contained representations for the evolution of spatial (3D) positions, as well as visualization and editing features. Each point of a trajectory has 3 Cartesian coordinates and an optional time-tag allowing to map the position to a temporal referential. These time-tags do not necessarily represent absolute time values and can possibly be modified (or generated) by additional processing (scaling, delays, sampling...) The trajectories generated in Figure 1–b, for instance, do not include specific time information and will be unfolded depending on the onsets and durations specified for their respective sources in the matrix scene representation. The trajectories also have some lazy properties through a number of parameters allowing to define how time unfolding is to be computed, given the explicit timed-points and some rules, for instance respecting a constant speed (hence depending on the distance between successive points), or assuming a constant time interval between the same successive points (and independently of their relative distance). Late sampling (or re-sampling) is also often useful in order to keep compositional specifications relatively small and easily controllable (using reduced sets of "control points") and to convert them to smoother movements and evolutions at rendering time.[3]

VI. Control of Spatial Rendering: OM-Spat

Due to a certain level of abstraction maintained in the specification of the spatial sound scenes during the early compositional stages, in principle no excessive specific knowledge is required about the rendering process, which is most often carried out using external software systems.[4]

The SDIF file format [39] is used as a standard to encode the spatial scene descriptions created in OM and transfer them

[1] These two approaches are not completely incompatible, and most observed applications and practices actually partly involve both the "abstract" spatial thinking and a consideration of particular targeted spatialization systems.

[2] We call "spatial sound scene" a set of sound sources associated to positions, trajectories, directivity patterns and other space- or acoustic-related features.

[3] The *path* object in [23] provides similar features for the specification of trajectories in Common Lisp Music [36].

[4] This "stratified" vision [29] is not always completely realistic in fact, since elements of the rendering techniques often need to be specified and incorporated at the control level.

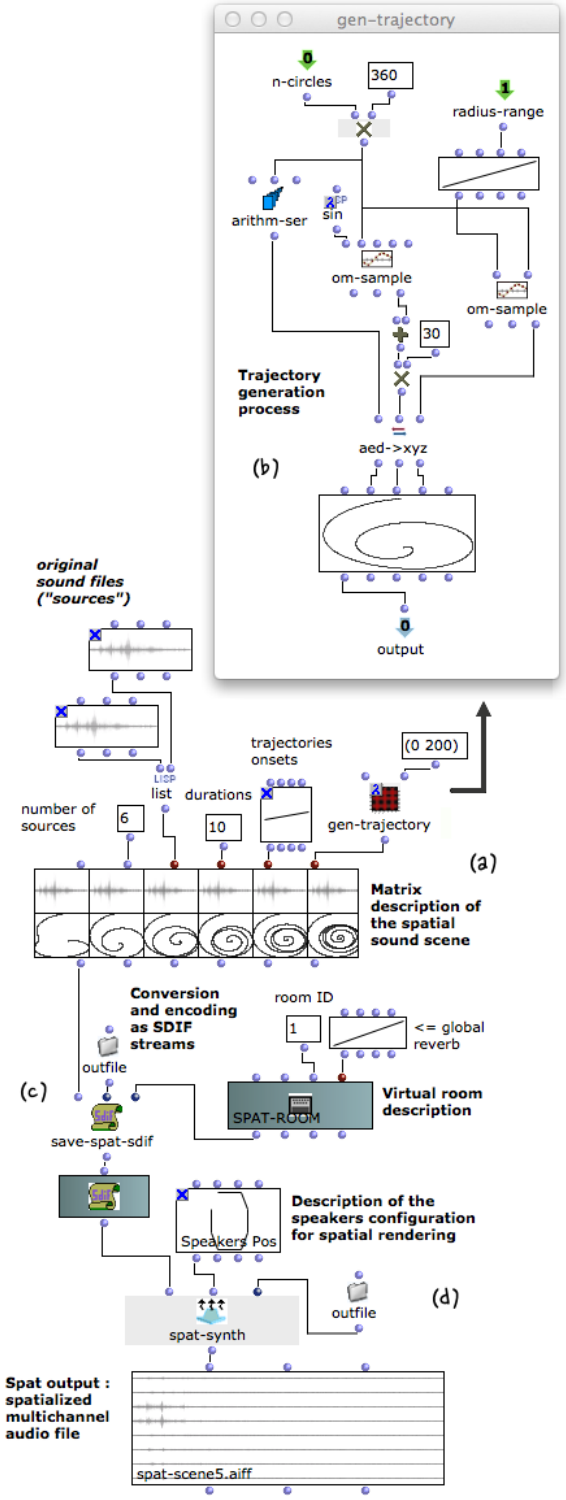


Fig. 1. Representation and rendering of a spatial sound scene in OpenMusic. The matrix representation of the scene (a) is instantiated from a set of sound sources and spatialization parameters provided either "literally" (e.g. durations, onsets...) or as functional specification (e.g. the trajectory generation process (b) provided as higher-order function). It is eventually encoded as an SDIF file (c), and here rendered to a multi-channel audio file (d).

to external software and rendering environments.[5] SDIF is a "polyphonic", stream-based format allowing to encode timed frames containing any kind of data structured as matrices. Each matrix (identified by a type) contains a number of components described at the time of the containing frame by a set of field values (e.g. x, y, z in the case of spatial positions). Several matrices of different types can coexist in common frames, and several frame streams can coexist in a same SDIF description file. SDIF is therefore quite well adapted to render the OM spatial sound scenes (see Figure 1–c): source descriptions can be can be interpreted in terms of such flat and precisely timed streams describing the evolution of their different spatial parameters (see [12]).

Specific matrix types have been defined in SDIF corresponding to the main control parameters of the Ircam *Spatialisateur*. A command line rendering tool developed from this software (*Spat renderer*[6]) allows to generate spatialized multichannel audio files from the SDIF descriptions created in OM (see Figure 1–d).

As shown in Figure 1, room descriptions can also be addressed in the compositional and spatialization processes: the *Spatialisateur* provides powerful perceptual room modelling features, to which can be "attached" the different sources.[7][8]

Both SDIF file conversion and *Spat* rendering features described in this section are available in the OpenMusic *OM-Spat* library.

VII. Data Streaming and Real-Time Integration

In the present musical and technological context, sound spatialization is either performed as a completely off-line process (most often, in the case of pure electronic music), or in real time during concerts and performances. In the latter case a lower degree of abstraction and complexity is affordable for the description, control and rendering of spatial sounds.

The streaming of the SDIF-formatted data generated in OM (hence already "flattened", timed and ordered) is envisaged as one solution in the integration of spatial data, prepared beforehand in the computer-aided composition environment, in external real-time processes.

Spat-SDIF-Player is a standalone application developed for this purpose (see Figure 2–a). Implemented in Max/MSP [32] using the MuBu buffering library [35], this application loads SDIF spatial description files and provides standard playback controls as well as additional features such as stream (i.e. source) selection, looping or speed control. Note that this player does not produce any sound but broadcasts control messages via UDP. The messages are formatted in OSC [44] and respect the SpatDIF specification [28] so that any

[5]SDIF is also used to interchange data between the OM compositional environment and external sound rendering or other lower-level processing tools [8].

[6]*Spat renderer* by Thibaut Carpentier, Ircam.

[7]Several simultaneous "virtual rooms" can coexist and be superimposed in a same spatial sound scene, attached to different sound sources.

[8]In the SDIF description, separate frame streams are used for the room parameters and their possible evolution through time, and a "room ID" attribute allow to match the sources to one of the (possibly multiple) rooms.

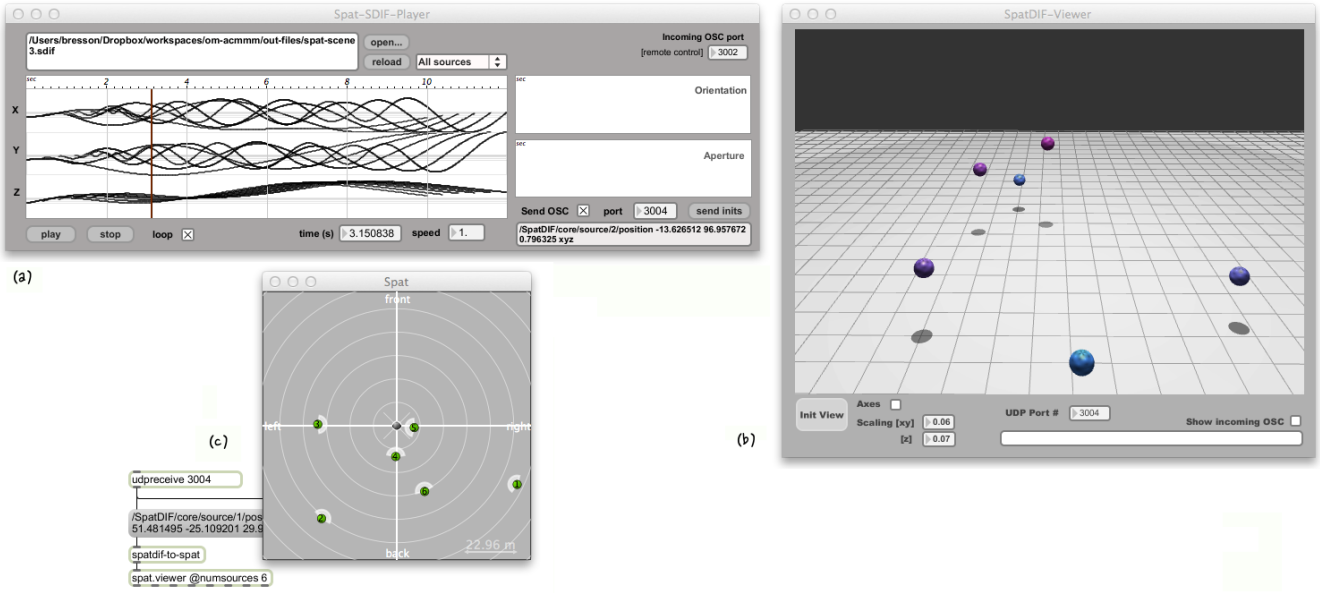


Fig. 2. Streaming of SDIF spatial descriptions using *Spat-SDIF-Player* (a). Control messages are be received and interpreted by the *SpatDIF-Viewer* 3D renderer (b) and by the *Spat* library tools in Max/MSP (c).

rendering system compliant with this format can interpret and eventually render the spatial descriptions accordingly. Figure 2 shows two such examples: one (*SpatDIF-Viewer*, Figure 2–b) is a visualization tool rendering SpatDIF messages in a 3D display, and the second one is a Max/MSP patch receiving the same data and converting them into control messages for the *spat.oper* object in the *Spat 4* library (Figure 2–c).

This networking protocol provides interesting flexibility and inter-operability between applications and tools for spatialization. It is however limited as messages may be numerous (for instance in case of high sample rates and/or when numerous spatialization parameters are involved), and are necessarily sent sequentially. In these cases, the simultaneous control of multiple sound sources (which is not a limitation at the level of the specification of the spatial sound scenes in OM, for instance) can raise synchronization issues, delays, or other undesired behaviours.

VIII. Spatial Sound Synthesis

An interesting concept developed and implemented in the context of these works on sound spatialization is the one of *spatial sound synthesis* [37]. Generalizing similar ideas developed with the spatialization of spectral or sinusoidal decompositions [22], [42], or with granular and swarm-based models [24], [43], the concept of spatial sound synthesis consists in addressing spatial issues at the very same level as the sound synthesis itself.

Sound synthesis processes can be represented and controlled in OpenMusic using matrix objects (in a similar way as is done with spatial sound scenes—see Section IV). In the *OMChroma* library [2], these matrices describe parameter values corresponding to a given sound synthesis "instrument", for a number of synthesis components (or virtual instances of this instrument). This representation in OM visual programs is eventually converted to textual code compiled into a sound file by the Csound synthesizer [7]. Typically, from a simple digital oscillator implemented in Csound and used as the "synthesis instrument", an *OMChroma* matrix would allow to describe and store the frequency, amplitude, duration and possible additional parameters of this instrument for an arbitrary (and possibly important) number of components, hence implementing a powerful "additive synthesis" control and rendering process (see Figure 3–a).

Synthesis processes in *OMChroma* can be extended to sound spatialization by devising appropriate Csound instruments considering sound sources as one of the inputs of the synthesis, and the multichannel output as its result. As with the matrices described in section IV, spatialization processes developed using *OMChroma* can therefore make for unlimited polyphony (number of sources), and provide an important diversity in the rendering techniques thanks to the numerous spatialization tools and programming possibilities available in the Csound language. The *OMPrisma* library, developed by Marlon Schumacher at CIRMMT (McGill University) is an extension of *OMChroma* providing a rich set of such spatialization options to be used in complement (or combination—see below) to the *OMChroma* synthesis objects [38].

More interestingly, it is possible to develop both the synthesis and spatialization processes in a same DSP instrument, and thereby to build arbitrarily complex sound structures including spatial descriptions for every single component of the sound synthesis. In the "additive synthesis" example given above, for instance, one could imagine to assign a specific trajectory to every single "partial" of the sound (or atomic sinusoidal component, hence considered as an individual sound

source). Unique and innovative sound textures can therefore be designed establishing strong relations between the sound synthesis parameters and corresponding spatial morphologies.

In order to allow for unconstrained combination between the different sound synthesis techniques provided in *OMChroma* with the spatial rendering tools implemented in *OMPrisma*, a "merging" protocol has been defined between synthesis and spatialization objects (and subsequently between corresponding Csound source code). This dynamic integration provides a high degree of flexibility for the experimentation on the different spatialization and synthesis techniques combinations; Any sound synthesis object can be connected and merged to any spatialization object in order to perform a spatial sound synthesis process integrating both parameters and attributes (see Figure 3).

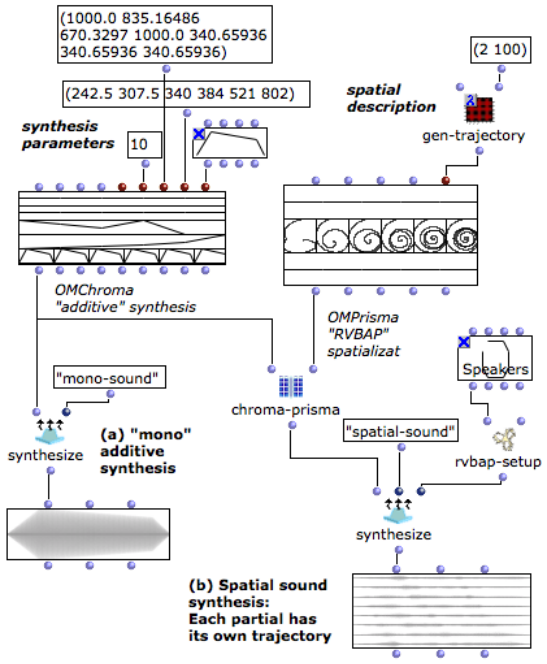


Fig. 3. Spatial sound synthesis in OM with *OMChroma*/*OMPrisma*.

IX. Conclusion: Toward Extended Computational Paradigms for Spatialization ?

Sound spatialization is now a widespread concern in electro-acoustic and contemporary music, and a major issue in computer music research and development. The technological advanced of the recent years opened a world of possibilities, and many musical venues and research institutions are now equipped with high-quality spatial rendering facilities.[9]

Compositional concerns and research currently emerge on top of these technologies, as show for instance the different tools and projects presented in this article, in OpenMusic and more generally in the computer-aided composition domain. An interesting point here is the fact that the spatialization processes can now be thought and carried out inside or in a close relation to the compositional processes and corresponding formalized approaches.

In this regard, the connection to real-time frameworks is probably still a critical—and interesting—aspect: while most of the signal processing tools render spatial sounds in real-time, their integration with compositional inputs and specifications is not straightforward (the works presented in Section VII are preliminary attempts in this direction). More and more frequently as high-resolution systems get developed and available, the integration of relatively complex sound structures and spatial control can raise computational issues which can be solved by merging off-line generation of compositional inputs, using dedicated computational paradigms and environments, to reactive (real-time) spatial sound rendering systems.

In this respect and in the continuation of the different past projects dealing with spatial concerns in computer-aided composition, interesting new directions could involve extended constraint-based approaches to spatial structures, or specific formal frameworks such as qualitative spatial reasoning [13]. By including spatial concepts in the programs generating and processing data, spatial computing [15] could also be a promising approach to the control of spatialization processes and to cope with the general issue of composing spatial structures using computer processes.

[9]Notable example of concert venues and research facilities providing high-quality spatialization environments include Ircam's *Espace de projection* with its WFS and higher-order ambisonics systems, the *Acousmonium* at the GRM (Paris), the Birmingham ElectroAcoustic Sound Theatre (*BEAST*), the *Allosphere* at UC Santa Barbara, the ZKM *Klangdom* in Karlsruhe, the *Sonic Lab* in Belfast, and many others.

Acknowledgements

Significant parts of the works presented in this paper have been carried out in collaboration with Marlon Schumacher at Ircam and at CIRMMT, McGill University. The author would also like to acknowledge Thibaut Carpentier for his collaboration on the *Spat* rendering and on the definition the interchange formats used in this project.

References

[1] C. Agon, G. Assayag and J. Bresson (Eds.), *The OM Composers Book. 1*, Editions Delatour/Ircam, 2006.

[2] C. Agon, J. Bresson and M. Stroppa, "OMChroma: Compositional Control of Sound Synthesis." *Computer Music Journal*, 35(2), 2011.

[3] G. Assayag, "Visual Programming in Music." *Proceedings of the International Computer Music Conference*, Banff Canada, 1995.

[4] G. Assayag, C. Rueda, M. Laurson, C. Agon and O. Delerue, "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic." *Computer Music Journal*, 23(3), 1999.

[5] M. A.J. Baalman, "Spatial Composition Techniques and Sound Spatialization Technologies." *Organised Sound*, 15(3), 2010.

[6] A. J. Berkhout, D. de Vries, D. and P. Vogel, "Acoustic Control by Wave Field Synthesis." *Journal of the Acoustical Society of America*, 93, 1993.

[7] R. Boulanger (Ed.) *The Csound Book: Perspectives in Software Synthesis, Sound Design, Signal Processing,and Programming*, The MIT Press, 2000.

[8] J. Bresson and C. Agon, "SDIF Sound Description Data Representation and Manipulation in Computer-Assisted Composition." *Proceedings of the International Computer Music Conference*, Miami, USA, 2004.

[9] J. Bresson, C. Agon and G. Assayag (Eds.), *The OM Composers Book. 2*, Editions Delatour/Ircam, 2008.

[10] J. Bresson, C. Agon and G. Assayag, "Visual Lisp/CLOS Programming in OpenMusic." *Higher-Order and Symbolic Computation*, 22(1), 2009.

[11] J. Bresson, C. Agon and G. Assayag, "OpenMusic. Visual Programming Environment for Music Composition, Analysis and Research." *Proceedings of ACM MultiMedia*, Scottsdale, USA, 2011.

[12] J. Bresson and M. Schumacher, "Representation and Interchange of Sound Spatialization Data for Compositional Applications." *Proceedings of the Inter- national Computer Music Conference*, Huddersfield, UK, 2011.

[13] A. G. Cohn and J. Renz, "Qualitative Spatial Reasoning," in F. Harmelen, V. Lifschitz and B. Porter (Eds.), *Handbook of Knowledge Representation*, Elsevier, 2007.

[14] J. Daniel, *Représentation de champs acoustiques, applications à la transmission et à la reproduction de scènes sonores complexes dans un contexte multimedia*, PhD Thesis, Université Pierre et Marie Curie, Paris 6, France.

[15] A. De Hon, J.-L. Giavitto and F. Gruau, *Computing Media and Languages for Space-Oriented Computation*, Dagsthul Seminar Proceedings, Dagsthul, 2006.

[16] O. Delerue, *Spatialisation du son et programmation par contraintes : Le système MusicSpace*. PhD Thesis, Université Pierre et Marie Curie, Paris 6, France, 2004.

[17] O. Delerue and C. Agon, "OpenMusic + MusicSpace = OpenSpace." *Actes des Journées d'Informatique Musicale*, Issy-les-moulineaux, France, 1999.

[18] M. Geier, M., J. Ahrens and S. Spors, "The SoundScape Renderer: A Unified Spatial Audio Reproduction Framework for Arbitrary Rendering Methods." *AES 124th Convention*, Amsterdam, The Netherlands, 2008.

[19] M. A. Harley, *Space and Spatialization in Contemporary Music: History and Analysis, Ideas and Implementations.* PhD Thesis, McGill University, Montreal, Canada, 1994.

[20] C. Jaksjø, "Architecture as Virtual Music (Re-Actualizing *Zonnestraal*)," in J. Bresson, C. Agon and G. Assayag (Eds.), *The OM Composers Book. 2*, Editions Delatour/Ircam, 2008.

[21] L.-M. Jot and O. Warusfel, "A Real-Time Spatial Sound Processor for Music and Virtual Reality Applications." *Proceedings of International Computer Music Conference*, Banff, Canada, 1995.

[22] D. Kim-Boyle, "Spectral Spatialization - An Overview." *Proceedings of the International Computer Music Conference*, Belfast, Ireland, 2008.

[23] F. Lopez-Lezcano, "A Dynamic Spatial Locator ugen for CLM." *Proceedings of the Sound and Music Computing Conference*, Berlin, Germany, 2008.

[24] A. McLeran, C. Roads, B. L. Sturm, J. J. Shynk, "Granular Sound Spatialization Using Dictionary-Based Methods." *Proceedings of the Sound and Music Computing Conference*, Berlin, Germany, 2008.

[25] G. Nouno, "Some Considerations on Brian Ferneyhough's Musical Language through His Use of CAC. Part II – Spatialization as a Musical Parameter," in J. Bresson, C. Agon and G. Assayag (Eds.), *The OM Composers Book. 2*, Editions Delatour/Ircam, 2008.

[26] G. Nouno and C. Agon, "Contrôle de la spatialisation comme paramètre musical." *Actes des Journées dInformatique Musicale*, Marseille, France, 2002.

[27] F. Otondo, "Contemporary Trends in the Use of Space in Electroacoustic Music." *Organised Sound*, 13(1), 2008.

[28] N. Peters, S. Ferguson and S. McAdams, "Towards a Spatial Sound Description Interchange Format (SpatDIF)." *Canadian Acoustics*, 35(3), 2007.

[29] N. Peters, T. Lossius, J. Schacher, P. Baltazar, C. Bascou and T. Place, "A Stratified Approach for Sound Spatialization." *Proceedings of the Sound and Music Computing Conference*, Porto, Portugal, 2009.

[30] N. Peters, G. Marentakis, S. McAdams, "Current Technologies and Compositional Practices for Spatialization: A Qualitative and Quantitative Analysis." *Computer Music Journal*, 35(1), 2011.

[31] L. Pottier, "Dynamical Spatialisation of sound. HOLOPHON: a graphical and algorithmical editor for Σ1. *Proceedings of the International Conference on Digital Audio Effects*, Barcelona, Spain, 2008.

[32] M. Puckette, "Combining Event and Signal in the MAX Graphical Programming Environment." *Computer Music Journal*, 15(3), 1991.

[33] V. Pulkki, "Virtual sound source positioning using vector base amplitude panning." *Journal of the Audio Engineering Society*, 45(6), 1997.

[34] J. C. Schacher and P. Kocher, "Ambisonics spatialisation Tools for Max/MSP." *Proceedings of the International Computer Music Conference*, New Orleans, USA, 2006.

[35] N. Schnell, A. Röbel, D. Schwarz, G. Peeters and R. Borghesi, "MuBu & Friends - Assembling Tools for Content Based Real-Time Interactive Audio Processing in Max/MSP." *Proceedings of the International Computer Music Conference*, Montreal, QC, Canada, 2009.

[36] B. Schottstaedt, "CLM: Music V Meets Common Lisp." *Computer Music Journal*, 18(2), 1994.

[37] M. Schumacher and J. Bresson, "Spatial Sound Synthesis in Computer-Aided Composition." *Organised Sound*, 15(3), 2010.

[38] M. Schumacher and J. Bresson, "Compositional Control of Periphonic Sound Spatialization." *2nd International Symposium on Ambisonics and Spherical Acoustics*, Paris, France, 2010.

[39] D. Schwartz and M. Wright, "Extensions and Applications of the SDIF Sound Description Interchange Format." *Proceedings of the International Computer Music Conference*, Berlin, Germany, 2000.

[40] E. Stefani and K. Lauke, "Music, Space and Theatre: Site-Specific Approaches to Multichannel Spatialization." *Organised Sound*, 15(3), 2010.

[41] T. Todoroff, C. Traube and J.M. Ledent, "NeXTSTEP Graphical Interfaces to Control Sound Processing and Spatialization Instruments." *Proceedings of the International Computer Music Conference*, Thessaloniki, Greece, 1997.

[42] D. Topper, M. Burtner and S. Serafin, "Spatio-Operational Spectral (S.O.S.) Synthesis." *Proceedings of the International Conference on Digital Audio Effects*, Hamburg, Germany, 2002.

[43] S. Wilson, "Spatial Swarm Granulation." *Proceedings of the International Computer Music Conference*, Belfast, Ireland, 2008.

[44] M. Wright, "Open Sound Control: An Enabling Technology for Musical Networking." *Organised Sound*, 10(3), 2005.

Decentralized spatial algorithm design

Matt Duckham

***Abstract*—Spatial computers present challenges to conventional distributed algorithm design. Substantive progress is being made in developing new algorithm design tools and techniques, for example in the development of the Proto language. This paper summarizes an alternative but complementary technique targeted at the specification of decentralized algorithms for spatial computing. The approach focuses on abstract, implementation-independent specification of designs, as opposed to more practical programming constructs. The aim is to speed the development and ease the communication of algorithms designs. This is achieved augmenting an established distributed algorithm design technique with the minimal additional constructs required to compute with diverse spatiotemporal objects and relationships. The paper illustrates the additional spatial and temporal structures using the running example of decentralized algorithms for spatial region boundary detection.**

I. Introduction

Spatial computing can be characterized as a special case of distributed computing, where additional *geographic* constraints to the generation and communication of information exist. The challenge set in [1] is "to conceive of how to reformulate [distributed systems] applications for a continuous geometric world."

This paper describes an approach to designing *decentralized spatial algorithms*. A decentralized system is a distributed system where no single system element possesses global knowledge of the system state [2]. Consequently, decentralized spatial algorithms are well-suited to spatial computing environments, which present geographic constraints to both the generation and movement of information. Our approach is complementary to, but distinct from related approaches in spatial computing, in particular [1], [3], [4]. In comparison with [1], [3], [4], our approach focuses more strongly on the algorithm design and specification. We augment an established distributed algorithm design procedure with the spatiotemporal structures required for decentralized computing with many different types of spatiotemporal objects, references, and relations. As a consequence, however, our approach does not focus so strongly on practical programming architectures and implementation—something that is an important focus and contribution of [1], [3], [4].

Following a brief review of related work, the established distributed algorithm design technique upon which our approach is founded is introduced (Section III, after [5]). We then identify, with examples, the fundamental spatial and temporal structures required for decentralized spatial algorithm design (Sections IV and V). Finally, before concluding (Section VII) the paper looks briefly at the role of agent-based simulation in algorithm design (Section VI).

M. Duckham is with the Department of Infrastructure Engineering, University of Melbourne, VIC 3010, Australia. E-mail: matt@duckham.org

II. Related work

As already highlighted, this work shares similar aims to research on the definition of languages for spatial computing, including the development of the Proto language [1], [3], [4], [6] as well as more broadly (see [7] for a survey). The design process summarized in this paper is, we believe, complementary to these efforts. Our approach favors assisting designers with the construction and communication of algorithms; but places less emphasis on practical implementation of these algorithms within spatial computers.

In attempting to construct an implementation-independent decentralized spatial algorithm design framework, it is essential to draw on established design tools and techniques. Hoare's CSP (communicating sequential processes [8]) and Robin Milner's CCS (calculus of communicating systems, [9]) are two examples of influential formal models that deal explicitly with the interactions between processes, and so are highly relevant. More recently, Milner's CCS has been extended with additional structure in the pi-calculus and bigraphs [10], [11].

These formalisms are being applied to fundamental problems in geographic information science (e.g., [12]), but are relatively complex to apply to higher-level domains, like algorithm design. Similarly, related models like IOA (input-output automata, [2]), have been applied to decentralized spatial algorithms (e.g., [13], [14]), but have a strong focus on proving formal properties, like fairness and liveness, rather than ease of construction or communication of designs.

So, while alternative models have the advantage of more formal rigor, their substantial additional complexity makes them less well-adapted to supporting the algorithm design process. Hence, in this paper we argue that the less formal but more intuitive technique of Nicola Santoro [5] provides a practical compromise between complexity and rigor.

III. Santoro's distributed algorithm design

The distributed algorithm design approach of [5] is founded on four key structures:

1) *Restrictions* on the environment in which the algorithm is designed to operate, such as restrictions on the network structure and connectivity, communication reliability and synchronization, and so forth.
2) System *events* that occur to nodes, specifically *receipt* of a message; *triggered* events (such as a scheduled alarm or periodic sensor reading); and a *spontaneous* impulse, external to the system.
3) *Actions* that a node can perform in response to the different events that occur—actions must be atomic sequences of operations that cannot be interrupted by other events.

4) *States* for nodes, which allow nodes to retain knowledge of previous interactions, and respond in different ways to the same event depending on the context.

For example, Algorithm 1 provides a simple decentralized algorithm in the style of [5] for identifying nodes at the boundary of a spatial region. With reference to the four key structures identified above:

Algorithm 1: Determining the (inner) region boundary

Restrictions: Reliable, fully asynchronous communication; undirected communication graph, $G = (V, E)$; sensor function $s : V \to \{0, 1\}$
State Trans. Sys.:
$(\{\text{INIT}, \text{IDLE}, \text{BNDY}\}, \{(\text{INIT}, \text{IDLE}), (\text{IDLE}, \text{BNDY})\})$
Initialization: All nodes in state INIT

INIT
  Spontaneously
    broadcast (`ping`, $\mathring{s}$) {Broadcast sensed value}
    become IDLE
  Receiving (`ping`, s')
    defer until IDLE

IDLE
  Receiving (`ping`, s')
    if $s' \neq \mathring{s}$ and $\mathring{s} = 1$ **then**
      become BNDY

- Restrictions: The algorithm makes no restrictions on synchronization between nodes (communication may be fully *asynchronous*), but does require *reliable* communication (messages sent will be received within some finite amount of time). Communication is assumed to be mediated through a bidirected communication graph G, but again no further restrictions on the communication graph structure are required. Finally, the algorithm does require a Boolean sensor on each node (capable of sensing either 1 or 0, e.g., in or out of a region of interest, "hot" or "cold," presence or absence of pollutant), represented as a sensor function $s : V \to \{0, 1\}$.
- States: The state transition system specifies at the beginning of the algorithm the defined states (INIT, IDLE, and BNDY) and allowable transitions (INIT to IDLE to BNDY). The initial states for all nodes are also specified in the algorithm header. The algorithm proper then defines for each state a (possibly empty) set of events and associated actions.
- Events and actions: Three events are defined in the algorithm. Nodes in the INIT state can spontaneously perform an action to broadcast a `ping` message, before transitioning to an IDLE state. Nodes in the IDLE state respond to `ping` messages received by checking if adjacent nodes sense a different value. If so, IDLE nodes transition into a BNDY state. Nodes in the INIT state receiving a `ping` message defer this event until the node is in the IDLE state (i.e., received message is placed on a stack and treated as received only after the node has transitioned into a IDLE state). Other possible events (e.g., receiving a `ping` message in the BNDY state) are by default associated with the empty action ("do nothing").

The intuition behind Algorithm 1 is that even without geometric information, based purely on communication neighborhoods, nodes can locally determine whether they are at a region boundary by comparing their local sensed data with that of their immediate one-hop neighbors.

While Algorithm 1 is kept deliberately simple, it does illustrate several key features of the approach. Most importantly, although the algorithm header specifies the *global* restrictions and states, the algorithm body only specifies *local* rules that each individual node executes in parallel. To enforce the rigid separation between local and global knowledge, we use the overdot notation to distinguish between a globally defined function, and an individual node's local knowledge about that function. For example, in the algorithm body, we write $\mathring{s}$ (read "my" s or "local" s) in place of $s(\circ)$, where $\circ \in V$ is the local node currently under consideration. A failure to adequately distinguish between the local information available to an individual node, and the global information available across the network is a major source of design errors in decentralized algorithms (e.g., writing an action for node v that attempts to access information that is not local to v, like $s(v')$).

In summary, the algorithm specification procedure adopted in this paper offers three main features:

- an established and standard toolkit for abstract and implementation-independent specification of decentralized algorithms, supporting improved communication between designers;
- an unambiguous specification of the computational and sensed environment in which the algorithm is designed to operate; and
- a rigid separation of local and global knowledge, helping to protect against design errors arising from incorrectly referring to inaccessible global information in a local protocol.

IV. Spatial extensions

The key question underpinning all of spatial information science is "What makes spatial special?" Similarly, extending the general distributed algorithm design technique in Section III to the special case of decentralized *spatial* algorithms requires the identification and selection of those characteristics that are "special" to spatial information.

Clearly, the most important spatial structure is location. However, "location" does not necessarily imply the *coordinate* location (usually termed *position*). Location may also involve a diversity of less detailed quantitative information about, for example, the relative distances or directions (bearings) between nodes, or even *qualitative* information about a node's proximity to other nodes or known locations.

Figure 1 summarizes six of the most common forms of location information. The six examples include: a. coordinate position with reference to some external coordinate framework, such as derived from GPS or virtual coordinate systems; b. relative *anchor* location, like proximity to some external

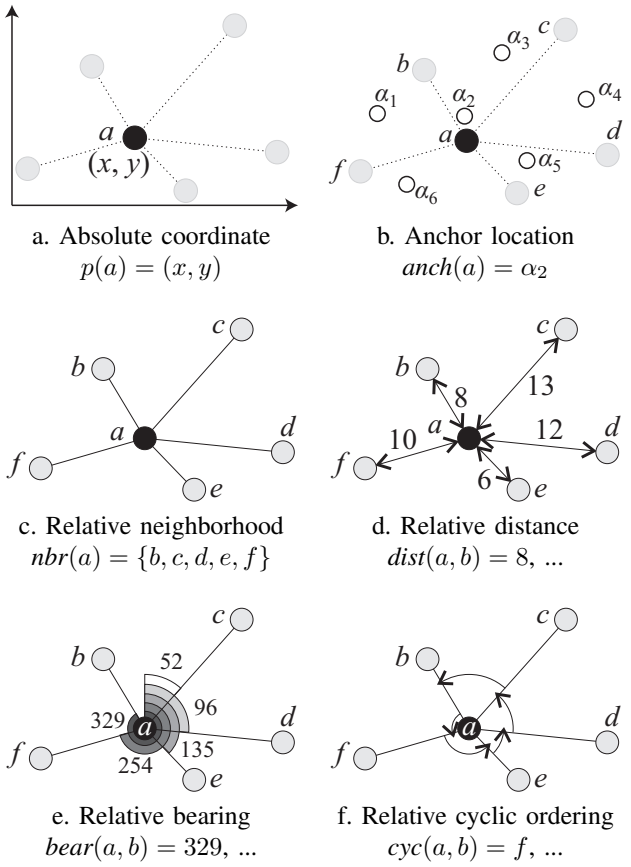


Fig. 1. Summary of six common types of location information available to a node.

"anchors" at known locations, such as might be derived from proximity-based RFID localization; c. relative neighborhood, such as knowledge of one-hop neighbors in a physical or overlay communication network; d. relative distance to neighbors, such as derived from range-finding technologies; e. relative bearing; and f. cyclic ordering, such as may be derived from direction-finding technologies.

The diversity of ways in which geographic location can be represented and related (e.g., cyclic ordering can be computed from bearings; coordinate position can be used to compute any of the other types of location information) is typical of problems in the spatial domain. Further information on localization techniques and technologies may be found in a range of literature on the topic, including [15]–[17].

Algorithm 2 provides an example extension to Algorithm 1 with more sophisticated spatial capabilities. The algorithm identifies not simply boundary nodes, but also a unique cycle of nodes around the region boundary. In practice, this requires each boundary node determine its next neighbor in the cycle, stored as local (i.e., to each node) data in the *wind* variable. Being able to cycle around a region boundary is a fundamental operation for a range of higher-level algorithms, such as computing the area or centroid of a region [18], testing containment between regions, efficient leader election for regions [19], or simply updating information stored at the region boundary [20]. Organizing communication around the boundary in this way is significantly more scalable than communicating over an entire spatial region [19].

Algorithm 2: Determining the (inner) boundary nodes and cycle for a region (cf. Algorithm 1).

Restrictions: Reliable, fully asynchronous communication; undirected planar communication graph, $G = (V, E)$; relative neighborhood, $nbr : V \to 2^V$; $s : V \to \{0, 1\}$; identifier function $id : V \to \mathbb{N}$; cyclic ordering $cyc : E' \to id_*$, where $E' = \{(v, id(v'))|(v, v') \in E\}$
State Trans. Sys.: $(\{\text{INIT}, \text{IDLE}, \text{BNDY}\}, \{(\text{INIT}, \text{IDLE}), (\text{IDLE}, \text{BNDY})\})$
Initialization: All nodes in state INIT
Local data: $wind : V \to V \cup \{\varnothing\}$, initialized $\mathring{wind} := \varnothing$; relation $D \subset \mathbb{N} \times \{0, 1\}$

INIT
  Spontaneously
    broadcast (ping, $\mathring{id}$, $\mathring{s}$)
    become IDLE
  Receiving (ping, i, d)
    defer until IDLE

IDLE
  Receiving (ping, i, d)
    set $D := D \cup (i, d)$ {Store id and sensed value}
    if $|D| = |\mathring{nbr}|$ **then** {Check if all ping received}
      Create function $data : I \to \{0, 1\}$, where $I = \{i'|(i', d') \in D\}$ and $data : i' \mapsto d'$
      if $\mathring{s} = 1$ and $0 \in data_*$ **then**
        set $\mathring{wind} := \mathring{cyc}(i'')$, where $data(\mathring{cyc}(i'')) = \mathring{s}$ and $data(\mathring{cyc}(i'')) \neq data(i'')$
        become BNDY

Ensuring a unique boundary cycle exists, and can be computed, requires: a. that the (overlay) network is planar[1]; and b. that nodes have access to local spatial information about the cyclic ordering of neighbors (listed in the restrictions to Algorithm 2). Information about the cyclic ordering of neighbors may be computed from geometric information, like absolute coordinates, or deduced via other means, such as direction finding. Irrespective of the details of the localization technology, it is possible to provide an abstract representation of the cyclic ordering as a function $cyc : E' \to id_*$, where $E' = \{(v, id(v'))|(v, v') \in E\}$. The function $id : V \to \mathbb{N}$ maps to an identifier for each node (such as hardware address), while id_* is the image of the *id* function (the set of identifiers mapped to by nodes). Making a clear distinction between a node itself, $v \in V$ (which cannot be communicated to neighbors), and the identifier of that node $id(v)$ (which can) is again important to accurate designs. It is never assumed that nodes have access even to immediate neighbors' states—

[1]Although simply stated, establishing and maintaining a planar network is often difficult in practice, for example due to positioning inaccuracies and environmental and energy fluctuations that affect network connectivity. Nevertheless, stating such restrictions in the algorithm header makes explicit and accessible those assumptions underlying an algorithm, assisting in subsequent comparison of different alternatives or in the context of specific deployment scenarios.

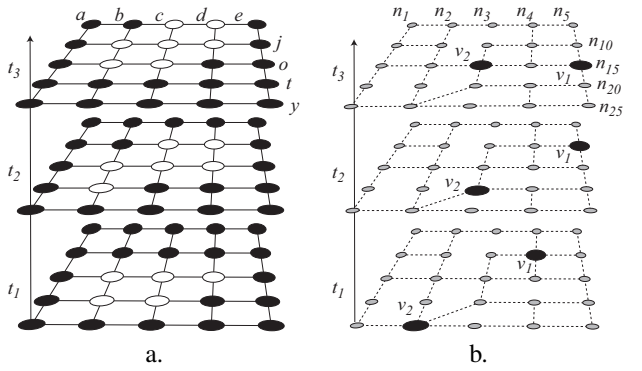


Fig. 2. Examples of spatiotemporal model of a. environmental dynamism ($s : V \times T \to \{black, white\}$) and b. node mobility ($anch : V \times T \to A$, where A is some set of known anchor locations, for example intersections in a transportation network).

all information that is not local to a node must be explicitly communicated to it by the algorithm before it can be used.

In summary, "spatial" involves more than coordinate position. Using a diversity of types of location information to efficiently construct higher-level spatial objects and relations, like boundaries and regions, groups and clusters, is a major challenge faced by decentralized spatial computing.

V. Temporal extensions

In a purely structural sense, time is a straightforward extension to our algorithm designs. Those functions that describe global restrictions to the algorithm can be easily augmented with an ordered set of discrete times T as part of their domain. For example, the atemporal positioning function $p : V \to \mathbb{R}^2$ can be extended to have time-varying positions as its domain, $p : V \times T \to \mathbb{R}^2$. Extensions to time-varying communication graphs can be similarly defined. In this way, both environmental dynamism and node mobility and volatility can be modeled (see Figure 2).

Algorithm 3 completes our boundary tracking example, showing an extension of the simple neighborhood-based boundary determination in Algorithm 1 to ongoing tracking of boundary status (this time through Boolean thresholding of a continuous sensor value, rather than a truly Boolean sensor). Currying allows time-varying functions to still be accessed locally. We adopt the database terminology *now* to indicate the current sensed value for a node (i.e., in Curried form $\mathring{s}(now)$).

Despite this apparent simplicity, dealing with time does introduce additional conceptual complexity into algorithms. A basic philosophical distinction is made usually between things that *endure* through time (called *endurants* or *continuants*), and things that *happen* in time (called *perdurants* or *occurrents*) [12]. Boundaries and regions are typical examples of geospatial endurants; the appearance, splitting, merging, and disappearance of regions are all examples of geospatial perdurants.

The distinction between endurants and perdurants maps directly to two fundamentally different types of information that may be generated by a decentralized spatiotemporal algorithm: *histories* and *chronicles* [21]. A history provides

Algorithm 3: Tracking the (inner) boundary of a region, with state maintenance.

Restrictions: Reliable, fully asynchronous communication; undirected planar communication graph $G = (V, E)$; $s : V \times T \to \mathbb{R}$; $id : V \to \mathbb{N}$; region threshold r

State Trans. Sys.: $(\{\text{INIT}, \text{IDLE}, \text{BNDY}\}, \{(\text{INIT}, \text{IDLE}), (\text{IDLE}, \text{BNDY}), (\text{BNDY}, \text{IDLE}), (\text{IDLE}, \text{INIT}), (\text{BNDY}, \text{INIT})\})$

Initialization: All nodes in state INIT

Local data: sensor reading at time of state change s_l; neighbor data $d : \mathring{nbr} \to \{-1, 0, 1\}$, initialized to $d(v) := -1$

INIT

Spontaneously

set $s_l := \mathring{s}(now)$ {Store last sensed value}

broadcast (`ping`, $\mathring{s}(now)$, $\mathring{id}$)

become IDLE

IDLE, BNDY

Spontaneously

if $\mathring{s}(now) = 1$ and $0 \in d_*$ **then**

become BNDY

else

become IDLE

Receiving (`ping`, s', i)

set $d(i) := s'$ {Store neighbor's sensed values}

When $\mathring{s}(now) < r \leq s_l$ **or** $s_l < r \leq \mathring{s}(now)$

become INIT

a spatiotemporal record of the states of monitored endurants (e.g., point locations, regions, boundaries, moving objects) through time. A chronicle provides a record of the occurrences (perdurants) that happened through time.

For example, imagine a spatial computer, like a sensor network, tasked with monitoring the spread of an oil spill (see Figure 3). We might wish the system to generate an alert when parts of the oil spill *appear* or *break up* (a chronicle). Alternatively, we might (also) wish the system to report on the *connectivity* of the oil spill every ten minutes over the course of a day (a history). We can expect to need to design decentralized spatiotemporal algorithms to monitor histories in some cases, and to monitor chronicles in other cases.

Thus, just as "spatial" involves more than simply coordinate location, so "temporal" involves more than simply timestamps; it means identifying and tracking salient spatial *events*.

VI. Algorithm simulation

Although our design approach aims to be implementation independent, we have developed a simulation system for implementing and evaluating decentralized spatial algorithm designs (see Figure 4), based on a popular agent-based simulation system called NetLogo [22]. A small number of additional keywords have been implemented as a NetLogo library, which make it possible to rapidly and directly translate pen-and-paper algorithm specifications into simulation models. A key advantage of using NetLogo is in its ability to simulate both decentralized spatial computing environments and dynamic

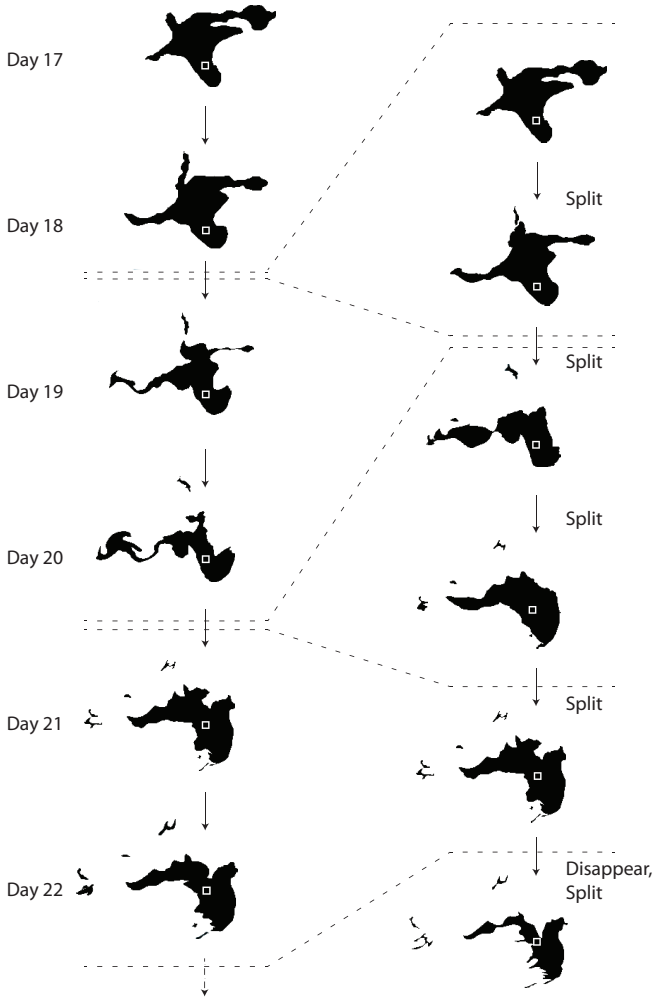


Fig. 3. Histories and chronicles: Two views of the changes in the extent of Deepwater Horizon Disaster oil slick, Gulf of Mexico, from Day 17, 7 May 2010. (Source: Times-Picayune).

geographic environments, supported by NetLogo's large and diverse community of domain scientist users (for example in ecology, biology, geography, and social sciences).

The ability to simulate algorithms can greatly assist the designer, by generating rapid feedback on algorithm behavior and as a basis for adversarial analysis to identify design flaws. As well as providing for empirical evaluation of the global behavior of decentralized spatial algorithms, such as scalability, simulations also can also help to explore experimentally the robustness of decentralized spatial algorithms. Spatial information is inherently uncertain, subject to inaccuracy (lack of correctness), imprecision (lack of detail), and vagueness (existence of borderline cases). In the case of spatial computing technologies, like sensor networks, inaccuracy and imprecision are especially important. Low cost, poorly calibrated sensors typically have relatively low accuracy; sensor observations are inevitably discrete in both space and time, the source of imprecision. Further, many of the application-level geographic objects and events of interest, like "hot spots" or "traffic jams," are vague (e.g., some location may be definitely

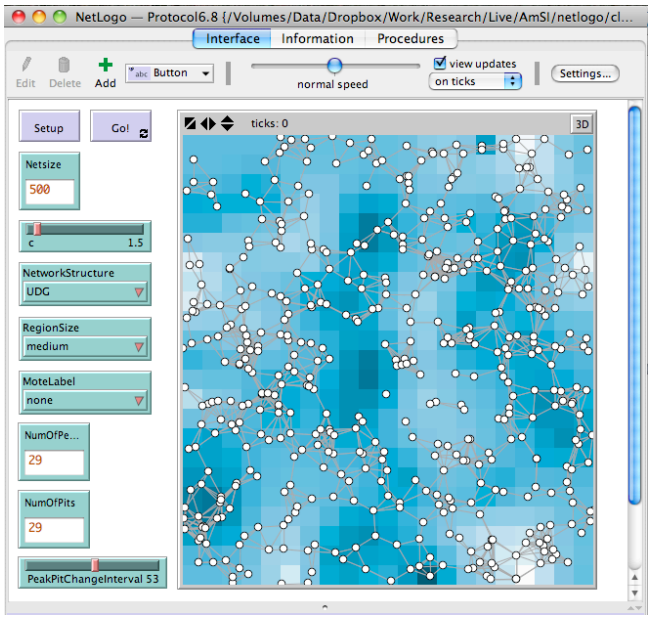


Fig. 4. Example NetLogo interface for decentralized spatial algorithm simulation

in a hot spot, others definitely not, but typically there will be borderline locations, for which it is indeterminate whether or not they are in or out of the hot spot). Decentralized spatial algorithms frequently need to demonstrate robustness to imprecision and inaccuracy in sensed information, and an ability to generate useful knowledge about vague geographic phenomena. Spatial computing under uncertainty is a key focus for current research.

VII. Summary

This paper has demonstrated how established distributed algorithm design techniques can be adapted to decentralized spatial algorithm design. The approach identifies a small number of spatial and temporal structures from which more sophisticated spatial computing algorithms can be constructed. Our approach complements and contrasts with existing research in [1], [3], [4], [6], and aims to help human designers in specifying local protocols that will exhibit the desired global behaviors. By contrast, [1], [3], [4], [6] target the (automated) transformation of global constructs into local protocols. Further, our approach emphasizes abstract, implementation-independent algorithm specifications, but does not explicitly address practical implementation and programming languages.

References

[1] J. Beal and R. Schantz, "A spatial computing approach to distributed algorithms," in *45th Asilomar Conference on Signals, Systems, and Computers*, 2010.

[2] N. Lynch, *Distributed Algorithms*. San Mateo, CA: Morgan Kaufmann, 1996.

[3] J. Bachrach, J. Beal, and J. McLurkin, "Composable continuous-space programs for robotic swarms," *Neural computing & applications*, vol. 19, no. 6, pp. 825–847, 2010.

[4] J. Bacharach and J. Beal, "Building spatial computers," Tech. Rep. 2007-017, MIT CSAIL, March 2007.

[5] N. Santoro, *Design and Analysis of Distributed Algorithms*. New Jersey: Wiley, 2007.

[6] J. Beal and J. Bachrach, "Infrastructure for engineered emergence in sensor/actuator networks," *IEEE Intelligent Systems*, pp. 10—19, 2006.

[7] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll, "Organizing the aggregate: Languages for spatial computing," in *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments* (M. Mernik, ed.), IGI Global, 2012, to appear. http://arxiv.org/abs/1202.5509.

[8] C. A. R. Hoare, "Communicating sequential processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666—677, 1978.

[9] R. Milner, *A Calculus of Communicating Systems*, vol. 92 of *Lecture Notes in Computer Science*. Springer Verlag, 1980.

[10] R. Milner, *Communicating and Mobile Systems: The π-calculus*. Cambridge University Press, 1999.

[11] R. Milner, "Pure bigraphs: Structure and dynamics," *Information and Computation*, vol. 204, pp. 60–122, 2006.

[12] M. Worboys, "Event-oriented aproaches to geographic phenomena," *International Journal of Geographic Information Science*, vol. 19, no. 1, pp. 1–28, 2005.

[13] M. Duckham and F. Reitsma, "Decentralized environmental simulation and feedback in robust geosensor networks," *Computers, Environment, and Urban Systems*, vol. 33, pp. 256–268, 2009.

[14] S. Dolev and N. Tzachar, "Empire of colonies: Self-stabilizing and self-organizing distributed algorithm," *Theoretical Computer Science*, vol. 410, pp. 514–532, 2009.

[15] J. Hightower and G. Boriello, "Location systems for ubiquitous computing," *IEEE Computer*, vol. 34, no. 8, pp. 57–66, 2001.

[16] M. Worboys and M. Duckham, *GIS: A Computing Perspective*. Boca Raton, FL: CRC Press, 2nd ed., 2004.

[17] H. Karl and A. Willig, *Protocols and Architectures for Wireless Sensor Networks*. Chichester, England: Wiley, 2005.

[18] M. Sadeq and M. Duckham, "Decentralized area computation for spatial regions," in *Proc. SIGSPATIAL ACMGIS*, (New York), pp. 432–435, ACM, 2009.

[19] M. Duckham, D. Nussbaum, J.-R. Sack, and N. Santoro, "Efficient, decentralized computation of the topology of spatial regions," *IEEE Transactions on Computers*, vol. 60, no. 8, pp. 1100–1113, 2011.

[20] M. J. Sadeq and M. Duckham, "Effect of neighborhood on in-network processing in sensor networks," in *Geographic Information Science* (T. Cova, K. Beard, M. Goodchild, and A. U. Frank, eds.), no. 5266 in Lecture Notes in Computer Science, pp. 133–150, Berlin: Springer, 2008. (Conference accepted 31% of submitted papers).

[21] A. Galton, "Fields and objects in space, time, and space-time," *Spatial Cognition and Computation*, vol. 4, no. 1, pp. 39–68, 2004.

[22] U. Willensky, "Netlogo." Center for Connected Learning and Computer-Based Modeling, Northwestern University, Evanston, IL., 1999. http://ccl.northwestern.edu/netlogo/.

The Evolution of Controller-Free Molecular Motors from Spatial Constraints

Jose David Fernández
Grupo de Estudios en Biomimética,
Departamento de Lenguajes y
Ciencias de la Computación,
Universidad de Málaga,
Málaga, Spain
josedavid@geb.uma.es

René Doursat
Grupo de Estudios en Biomimética,
Departamento de Lenguajes y
Ciencias de la Computación,
Universidad de Málaga,
Málaga, Spain
doursat@geb.uma.es

Francisco J. Vico
Grupo de Estudios en Biomimética,
Departamento de Lenguajes y
Ciencias de la Computación,
Universidad de Málaga,
Málaga, Spain
fjv@geb.uma.es

***Abstract*—Locomotion of robotic and virtual agents is a challenging task requiring the control of several degrees of freedom as well as the coordination of multiple subsystems. Traditionally, it is engineered by top-down design and fine-tuning of the agent's morphology and controller. A relatively recent trend in fields such as evolutionary robotics, computer animation and artificial life has been the *coevolution* and mutual adaptation of the morphology and controller in computational agent models. However, the controller is generally modeled as a complex system, often a neural or gene regulatory network. In the present study, inspired by molecular biology and based on normal modal analysis, we formulate a behavior-finding framework for the design of bipedal agents that are able to walk along a filament and have *no explicit control system*. Instead, agents interact with their environment in a purely reactive way. A simple mutation operator, based on physical relaxation, is used to drive the evolutionary search. Results show that gait patterns can be evolutionarily engineered from the spatial interaction between precisely tuned morphologies and the environment.**

***Index Terms*—morphological computation, elastic network model, behavior-finding**

I. Introduction

Engineers have made remarkable progress in their ability to design complex products. However, current engineering practice still favors a top-down approach, where the main problem is manually divided into smaller ones in order to maintain the overall complexity reasonably manageable. This procedure is rather loosely defined and ultimately relies on human expertise and creativity, which are skills that typically involve high costs, are unreliable and are difficult to formalize. Moreover, the ever increasing complexity of current engineered systems and robustness requirements is reaching the feasibility limits of the current paradigm, forcing the implementation of new engineering methodologies.

Inspired by the biological evolution and morphogenesis of organisms, recent advances in the discipline of evolutionary computation propose a radically different approach. Genetic algorithms combined with artificial development mechanisms operate over a population of individuals encoded by *genomes* that govern a morphogenetic process producing self-constructed designs [1]. That is, the genome is not a blueprint of the design, but the set of instructions that indirectly build it. The evolutionary operations (mutations and crossover) are applied to the developmental generative process that build the design, not to the design itself. This approach has been shown to overcome the issues of scalability, adaptability, and evolvability present in traditional evolutionary algorithms (based on a genomic representation that encodes the design in a explicit way) when applied to complex problems [2]. As a result, evolutionary developmental algorithms have been tried in a wide range of design problems, including the structure and controller of robots [3], digital creatures in Artificial Life studies [4, 5, 6, 7, 8], and computer animated characters [9]. In almost all models, however, the control system is fairly complex (often based on some kind of recurrent neural network), and in many cases, we believe, unnecessarily so.

In a seminal work [10], Chandana Paul demonstrated that a whole body-control system is able to perform more complex computations than the control system alone. This observation spawned the concept of *morphological computation*—a design methodology for robotic-like agents to exploit the dynamics of interaction between the body and the control system, resulting in minimal control systems. Several applications have been proposed in the field of robotics, including the design of semipassive bipedal robots [11], tensegrity robots whose complex, coupled non-linear dynamics are harnessed to generate a gait pattern with minimal control [10], path-following agents [12], and robots with open-loop control systems and minimal numbers of degrees of freedom that can self-stabilize into fast gait patterns and generate diverse behaviors through the interaction between body and control system [13].

We present here a framework to generate bipedal agents that can walk along a filament, taking inspiration from biological molecular motors. Toward this goal, we apply a simple evolutionary heuristic based on normal modal analysis [14] and a behavior-finding strategy [12]. Our work can be construed as "morphological computation" in two ways. First, the behavior of the agents is not driven by a complex, network-based control system, but emerges from their spatial characteristics. Second, we use a simple and explicit genetic representation, combined with a physics-based mutation operator able to

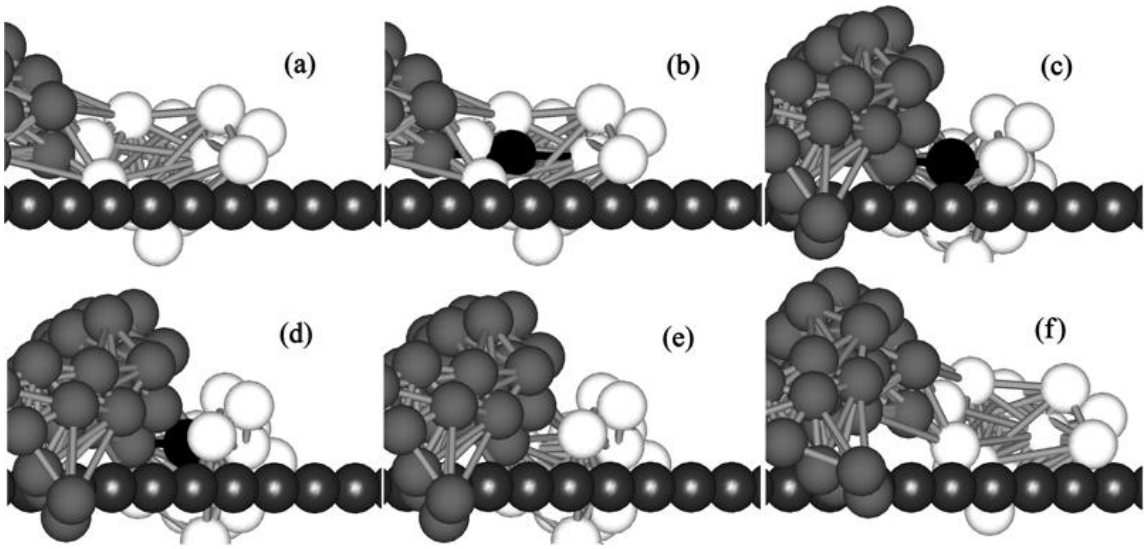


Figure 1. Working cycle of a motor template.

induce *coordinated changes* in the agents' structure. In this way, we take full advantage of the spatial and geometric nature of the genotype.

II. Methods

The model is motivated by biological molecular motors, such as the enzymes myosin, kinesin and dynein, capable of transforming chemical energy into mechanical work. Breaking down *ATP* molecules for power, these molecular motors can effectively *walk* along cellular filaments [15]. They are composed of one or two *motor heads*, each comprising a *catalytic core* (the site where ATP molecules attach) and a *docking site* (the site where the motor attaches to the filament). Each motor head undergoes a cycle (working cycle) of shape changes (*conformational changes*), powered by the energy stored in ATP molecules. The docking site cyclically attaches and detaches from the filament in a coordinated way, allowing the motor to advance through the filament.

Molecular motors can be construed as nanoscale robotic agents. The control system is implicitly defined in the specific biochemical interactions between the molecular motor, the ATP molecules, and the filament; in this way, their morphologies canalize the movements and the function of the motors [16]. Indeed, molecular motors represent a clear example of morphological computation. Taking inspiration from this observation, we have built a framework based on evolutionary optimization to design robotic agents that function in a way similar to molecular motors. We call these agents *motor templates*, following our earlier work on this topic [17]. A motor template represents the structure of a plausible protein. It is modeled by a folded chain of vertices, in which elastic links are established between two vertices if and only if their distance is less than a given threshold [18]. Thus the whole object constitutes a 3D mass-spring network. While modeling molecular motors with mass-spring networks may seem simplistic, it can be justified theoretically: for most proteins, including many molecular motors, the dynamics is mainly dictated by their overall structure rather than their specific biochemical compositions [19].

A. Motor templates

A template has two motor heads, each one endowed with a catalytic core and a docking site. The catalytic core is defined as a set of two nodes in the network. When an ATP molecule binds to the core, it is placed exactly in the middle of the two vertices, connected by a spring to each vertex in the pair. These springs are stretched to model the change in potential energy brought by the ATP molecule (this mechanism has been used in other studies, as [18]). The docking site is modeled as a set of nodes that can attach and detach from the filament. The working cycle of a motor head can be described as a reactive finite-state machine with four states:

1) *Sticky* state: The docking site is not in contact with the filament, and the catalytic core is empty (Figure 1a). This state ends when any node of the docking site touches the filament: the node becomes fixed to the filament, and an ATP molecule is bound to the catalytic core with stretched springs (Figure 1b). Then, the motor head transitions to the next state.
2) *Bound* state: The stretched springs introduced in the transition to this state induce a conformational change (Figure 1c), while the docking site remains firmly attached to the filament, resulting in a conformational change. After a fixed amount of time passes, the motor head transitions to the next state.
3) *Nonsticky* state: the nodes of the docking site detach from the filament, but remain in contact with it. If the activity of the other motor head or residual elastic forces drive the docking site out of contact with the filament (Figure 1d), the ATP is expulsed from the catalytic core, deleting the associated springs (Figure 1e). Then, the motor head transitions to the next state.

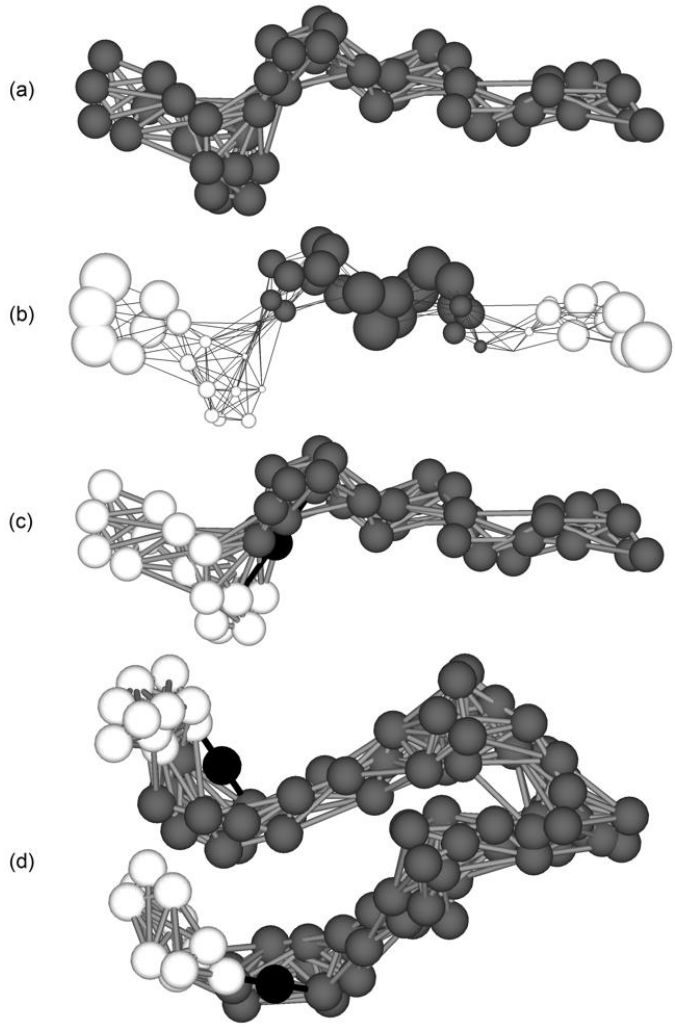


Figure 2. A mass-spring network (a) is processed to determine its catalytic cores and docking sites. The normal mode associated to the third eigenvector of its Kirchhoff matrix is shown (b). Each vertex is associated to a component of the eigenvector, whose magnitude (size) and sign (white positive, gray negative) conveys information about the vibration of the vertex in that normal mode, splitting the structure into three clusters. A motor head (c) is then composed of a catalytic core (ATP and connecting springs shown in black) placed between a distal cluster and the central one, and a docking site (in white). Finally, a motor template (d) is defined by joining two structures that are mirror images of each other.

4) *Relaxing* state: When the catalytic core becomes empty, the absence of the associated springs triggers another conformational change. After a fixed amount of time passes, the vertices of the docking site regain the ability to get fixed to the filament, and the motor head transitions to the initial state (Figure 1f), completing the cycle. A motor head has *completed* a working cycle when it has passed through all states and is back to the initial one: 1-2-3-4-1.

Simple and elegant theoretical tools that consider proteins as mass-spring networks, such as the Gaussian Network Model (GNM), use normal mode analysis to predict their structural and dynamical properties, and can do so to a surprising extent, including their unfolding pathways [20], their domain decomposition [21], and, in particular, their conformational changes and the position of their catalytic cores [22]. We use a heuristic based in GNM to determine the placement of the docking sites and catalytic cores, which are indirectly encoded in the morphology of the structure. Specifically, to define a motor head (with a catalytic core and a docking site) in the mass-spring network of a template (Figure 2a), we segment the network using the normal mode associated to the third eigenvector of its Kirchhoff matrix [22]. This eigenvector assigns a vibrational amplitude to each node in the network, which can be either positive or negative. In Figure 2b, each node's size and color represent the amplitude and sign, respectively (white is positive, gray is negative). Grouping neighboring nodes with same-sign vibrational amplitudes, three clusters can be defined in most mass-spring networks. There are two interfaces (hinges) between the clusters, such that two of the clusters are distal while the other one is central. As the third eigenvector is associated to a low-frequency normal mode, the interfaces heuristically indicate the places where the structure may bend easily in a conformational change [22]. In one of the interfaces, we introduce a catalytic core defined as a pair of nodes where ATP can bind (in Figure 2c, the ATP and its binding springs to the nodes of the core are shown in black), one node in a distal cluster and the other in the central one. As many pairs of vertices may exist, a heuristic is applied to select one of them. The docking site associated to the catalytic core is defined as the nodes of the associated distal cluster (Figure 2c, white nodes). Finally, the template is constructed by joining two instances of the structure (one of them the mirror image of the other) at the level of the first vertex in the chain of vertices, and setting a motor head at the opposite end of the structure (Figure 2d). This is inspired in the fact that many molecular motors function as dimers [15], i.e., they are composed of two joined identical proteins, each equipped with a motor head at their other extremity.

B. Evolutionary search

The genotype-phenotype mapping is direct at the morphological level: the genome *is* the 3D structure. At the functional level, however, the configuration of the motor heads is indirectly encoded by the structure, as described in the previous subsection.

To start an evolutionary optimization, the agents in the initial generation are generated as randomly folded chains of 50 nodes, defining relaxed springs between all neighboring nodes. Then, agents are evaluated in the following simulation: they are placed above a straight filament (made of nodes of the same size as the nodes of the structure), such that both docking sites touch it. One of the motor heads is set in the *sticky* state, while the other is set in the beginning of the *relaxing* state. If the structure and the configuration of the motor heads is adequate, coordinated working cycles (that is to say, their states change in a coordinated and cyclic way). After a preset amount of time passes, the simulation is stopped and the fitness is calculated to be the displacement of the agent's

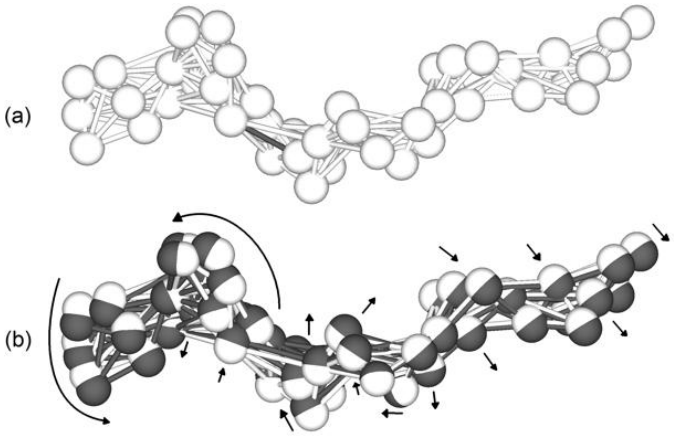


Figure 3. A mass-spring network structure is mutated (a) by enlarging the rest length of a spring (dark gray). The resulting structure after relaxation is shown (b) along with the original structure, in dark gray. Arrows point towards the main direction of displacement in each part of the structure.

center of mass in the direction of the filament, plus the number of completed working cycles by both motor heads.

For some structures, the heuristic cannot properly define the configuration of the motor heads (docking sites and catalytic cores). In this case, they are tagged as *nonevaluable* and are not subject to selection (they are eliminated from the evolutionary competition).

After the evaluation is done, a new population of agents is generated from the previous one by preferentially selecting agents with higher fitness. Finally, the mutation operator is applied (Figure 3). As the genotype-phenotype mapping is direct at the morphological level, the mutation operator must be able to bring many coordinated changes to the structure, in order to be effective. This can be accomplished by using a physics-based mutation: as each network is a spatial configuration of vertices connected by springs in resting state (neither compressed nor stretched), a mutation consists of changing the rest length of one or several springs, each one by an independent, random amount. These perturbations introduce potential energy in the mass-spring network. If it is allowed to relax through a physics simulation, the relative positions of many vertices will change in a coordinated manner (just as originally intended) to relieve the stress. After the relaxation process, the rest lengths of the springs are set to the new distances between nodes, and springs may be added (resp. deleted) if nodes become (resp. cease to be) neighbors. In each evolutionary run, a population of 100 templates undergoes the evaluation-selection-mutation cycle for 200 generations.

III. Results

The model has been tested in 38 evolutionary runs. In each run, 100 random mass-spring networks were generated to compose the corresponding initial population, 3800 in total. Almost all of them either walked a negligible distance or were nonevaluable (Figure 4). However, taking as a reference the distance walked by the best individual in each evolutionary run, significantly improved individuals have evolved, too. In

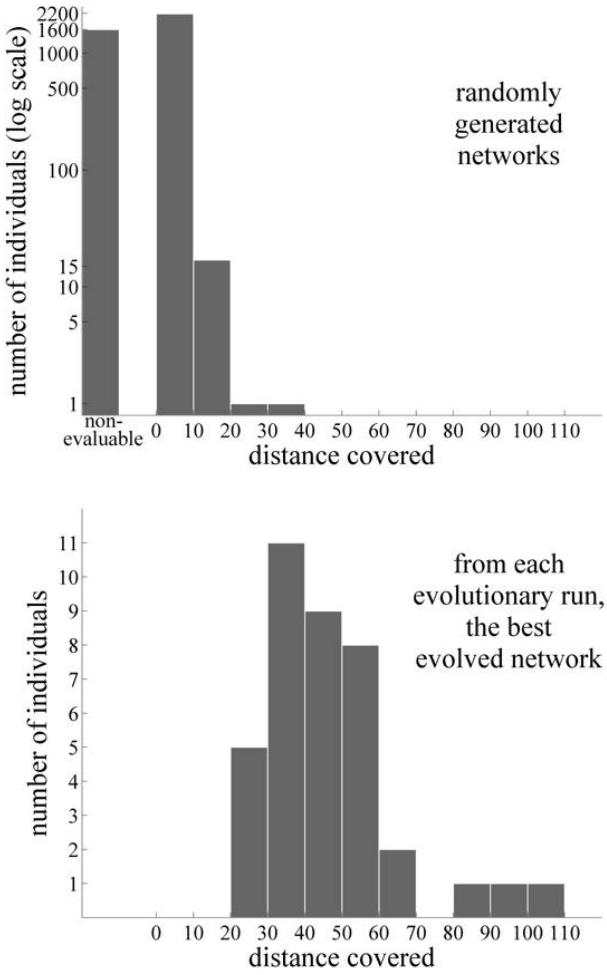


Figure 4. Histograms comparing the performance of 3800 randomly generated templates and the best evolved templates in 38 evolutionary runs. In the first histogram, a significant fraction of the templates ($\simeq$ 1600) are nonevaluable.

many cases, relatively minor modifications to the mass-spring network triggered a significant increase in the distance covered by the corresponding motor templates, suggesting that good templates needed to be precisely tuned to the working cycle and the details of the simulation.

The evolved bipedal templates feature a range of shapes and gaits:

- Walking *pseudo-legs* (Figure 5a) take short and secure alternate steps. The example shown here presents the peculiarity that the legs get attached to the filament at different angles, yet they still produce a steady gait.
- Slow, well-secured *pullers* (Figure 5b) keep a firm grip on the filament. Note that the limbs grasp the filament from below, while they join above it. This example rotates around the filament as it moves along it.
- *Hoppers* (Figure 5c) thrust themselves with both motor heads in an alternate way, only occasionally attaching both legs simultaneously to the filament. In the example provided here, the greater parts of the limbs are entan-

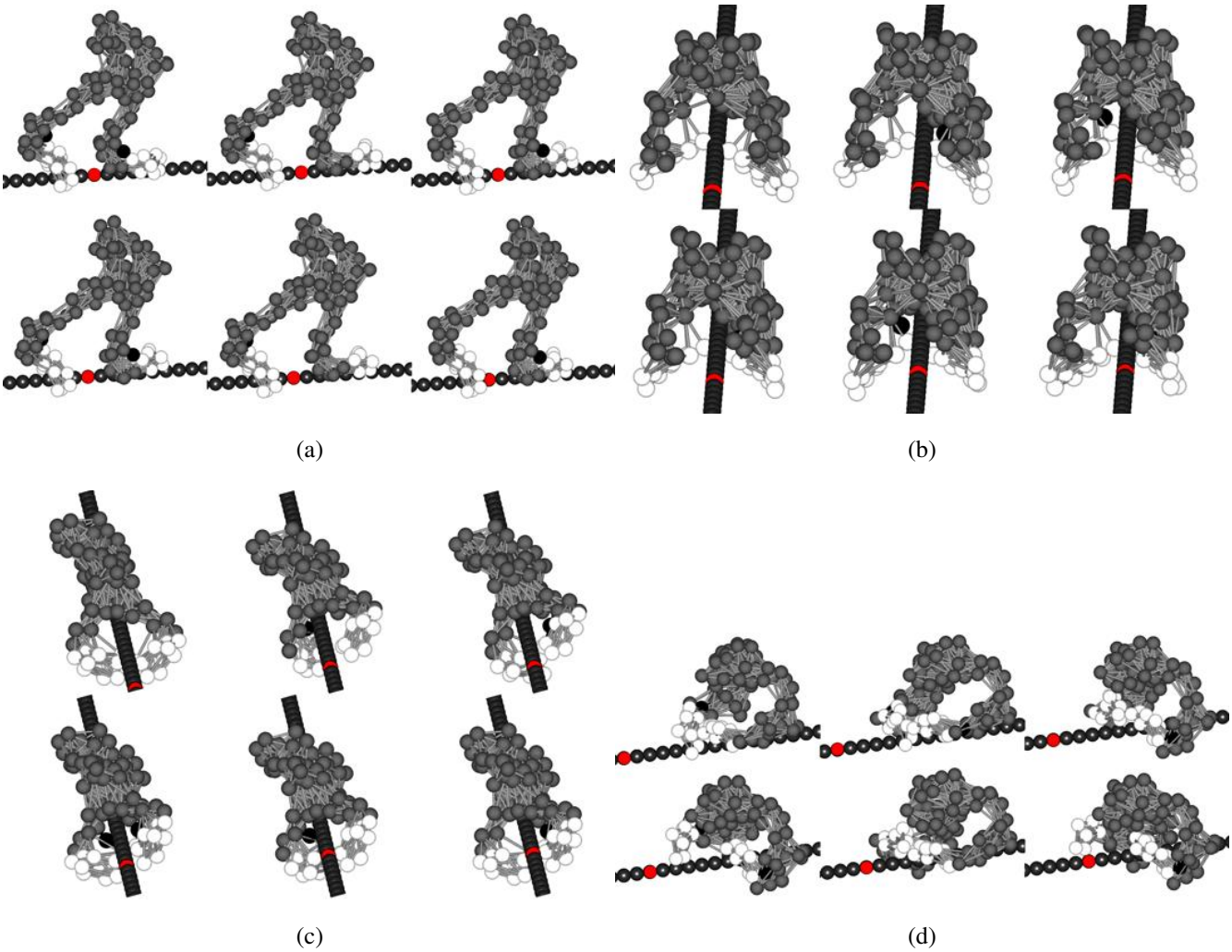


Figure 5. Sequences of snapshots illustrating the gait patterns of four evolved motor templates (in each case from left to right and from top to bottom). A node in the filament is marked in red to provide a point of reference.

gled into a single mass, effectively acting as cargo, and transported by comparatively small actuating limbs.

- Short but fast pulling *pseudo-limbs* (Figure 5d) are the fastest bipedal templates evolved in these experiments. This example has the peculiarity that the phase difference between both legs drifts in time.

IV. Discussion

In this study, we have presented a framework to generate motor templates (walking bipedal agents) inspired by biological molecular motors. The methodology consists of deriving the function of the agents from their structures (based on normal modal analysis), via a simple evolutionary algorithm and a physics-based mutation operator. The resulting structures can be interpreted as models of robotic agents made of elastic materials, suspended in a viscous fluid, while the "ATP molecules" that power the agents can be interpreted as simple actuators modifying the length of isolated parts of the structure.

As the structures are optimized to solve a functional problem (move forward as fast as possible) without morphological specifications, the problem can be described as *behavior-finding* [12] structure or morphology according to a set of constraints. The application of evolutionary optimization to behavior-finding tasks often yields diverse and sometimes unexpected solutions [12].

Many aspects of the model were specifically designed to be as simple as possible. The genome is minimal: it is only a fixed-width sequence of nodes in 3D space with springs between neighboring nodes, and the evolutionary algorithm is also very simple, including a single mutation operator and no crossover. Viable gait patterns could still be found in a high-dimensional space because the search was canalized in two ways:

- The working cycle (a simple reactive model) is hard-wired, and the configuration of the motor heads is indirectly encoded in the morphology of the agent.
- The mutation operator is based on physical relaxation after the application of perturbations to the structure, so it induces a fitness landscape that is more correlated to

the physical characteristics of the structure, which plays a key role in the configuration of gait patterns.

However, these features of the model are relatively low-level and did not constrain in any precise way the gait patterns of the templates. Thus the diversity of shapes and gait patterns was only enabled, not determined, by these characteristics and by the fact that the individuals competed in a 3D virtual world, coevolving their morphologies and behaviors (gait patterns). Morphogenesis arose by repeated application of a complex mutation operator through evolutionary time, instead of leveraging a complex genotype-phenotype mapping. As an example of morphological computation, gaits lacked any specific control subsystem: gait patterns emerged from the interaction between the properties, the physics, and the geometry of the templates and filament.

The mutation operator can also be considered as a mode of morphological computation. Instead of using heuristics based on the analysis of the characteristics of the structures, the mutation operator only perturbed the rest length of one or more springs in the structure. The new structure was then calculated by simulating physical relaxation, which naturally induced many coordinated changes into the mutated structure.

REFERENCES

[1] K. Stanley and R. Miikkulainen, "A taxonomy for artificial embryogeny," *Artificial Life*, vol. 9, no. 2, pp. 93–130, 2003.

[2] G. S. Hornby and J. B. Pollack, "Creating high-level components with a generative representation for body-brain evolution," *Artificial Life*, vol. 8, no. 3, pp. 223–246, 2002.

[3] D. Floreano and L. Keller, "Evolution of adaptive behaviour in robots by means of Darwinian selection," *PLoS Biology*, vol. 8, no. 1, p. e1000292, 01 2010.

[4] J. C. Bongard and R. Pfeifer, "Repeated structure and dissociation of genotypic and phenotypic complexity in artificial ontogeny," in *Proceedings of the 3rd Genetic and Evolutionary Computation Conference (GECCO)*. Morgan Kaufmann, 2001, pp. 829–836.

[5] G. S. Hornby, H. Lipson, and J. B. Pollack, "Generative representations for the automated design of modular physical robots," *IEEE Transactions on Robotics and Automation*, vol. 19, no. 4, pp. 703–719, 2003.

[6] M. Komosinski and A. Rotaru-Varga, "Comparison of different genotype encodings for simulated 3D agents," *Artificial Life*, vol. 7, no. 4, pp. 395–418, Fall 2001.

[7] J. B. Pollack, H. Lipson, G. Hornby, and P. Funes, "Three generations of automatically designed robots," *Artificial Life*, vol. 7, pp. 215–223, June 2001.

[8] L. Spector, J. Klein, and M. Feinstein, "Division blocks and the open-ended evolution of development, form, and behavior," in *Proceedings of the 9th Genetic and Evolutionary Computation Conference (GECCO)*. ACM, 2007, pp. 316–323.

[9] Y.-S. Shim and C.-H. Kim, "Generating flying creatures using body-brain co-evolution," in *Proceedings of the Symposium on Computer Animation (part of the 30th SIGGRAPH)*, ser. SCA '03. Aire-la-Ville, Switzerland, Switzerland: Eurographics Association, 2003, pp. 276–285.

[10] C. Paul, "Morphological computation: a basis for the analysis of morphology and control requirements," *Robotics and Autonomous Systems*, vol. 54, no. 8, pp. 619–630, 2006.

[11] K. Matsushita, M. Lungarella, C. Paul, and H. Yokoi, "Locomoting with less computation but more morphology," in *Proceedings of the 22nd IEEE International Conference on Robotics and Automation.*, April 2005, pp. 2008–2013.

[12] D. Lobo, "Evolutionary development based on genetic regulatory models for behavior-finding," Ph.D. dissertation, Universidad de Malaga, 2010.

[13] R. Pfeifer, F. Iida, and G. Gomez, "Morphological computation for adaptive behavior and cognition," *International Congress Series*, vol. 1291, pp. 22–29, June 2006.

[14] A. J. Rader, C. Chennubhotla, L.-W. Yang, and I. Bahar, *The Gaussian Network Model: theory and applications*. Chapman & Hall/CRC, 2006, ch. 3, pp. 41–63.

[15] M. Schliwa and G. Woehlke, "Molecular motors," *Nature*, vol. 422, no. 6933, pp. 759–765, April 2003.

[16] R. D. Vale and R. A. Milligan, "The way things move: looking under the hood of molecular motor proteins," *Science*, vol. 288, no. 5463, pp. 88–95, April 2000.

[17] J. D. Fernández and F. J. Vico, "Automating the search of molecular motor templates by evolutionary methods," *Biosystems*, vol. 106, pp. 82–93, 2011.

[18] Y. Togashi and A. S. Mikhailov, "Nonlinear relaxation dynamics in elastic networks and design principles of molecular machines," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 104, no. 21, pp. 8697–8702, May 2007.

[19] M. Lu, "The role of shape in determining molecular motions," *Biophysical Journal*, vol. 89, no. 4, pp. 2395–2401, October 2005.

[20] J. Su, "Protein unfolding behavior studied by elastic network model," *Biophysical Journal*, vol. 94, no. 12, pp. 4586–4596, June 2008.

[21] S. Kundu, D. C. Sorensen, G. N. Phillips, and Jr, "Automatic domain decomposition of proteins by a Gaussian Network Model," *Proteins: Structure, Function, and Bioinformatics*, vol. 57, no. 4, pp. 725–733, December 2004.

[22] L.-W. W. Yang and I. Bahar, "Coupling between catalytic site and collective dynamics: a requirement for mechanochemical activity of enzymes." *Structure*, vol. 13, no. 6, pp. 893–904, June 2005.

Arbitrary Nesting of Spatial Computations

Antoine Spicher*, Olivier Michel*, Jean-Louis Giavitto†

*LACL, Université Paris-Est Créteil,
61 av. du Général de Gaulle 94010 Créteil, France
Email: {olivier.micher,antoine.spicher}@u-pec.fr

†UMR 9912 STMS, Ircam & CNRS, UPMC, Inria
1 place Igor Stravinsky, 75004 Paris, France
Email: jean-louis.giavitto@ircam.fr

***Abstract*—Modern programming languages allow the definition and the use of *arbitrary* nested data structures but this is not generally considered in unconventional programming models. In this paper, we present arbitrary nesting in MGS, a spatial computing language. By considering different classes of neighborhood relationships, MGS can emulate several unconventional computing models from a programming point of view. The use of arbitrary nested spatial structures allows a hierarchical form of coupling between them. We propose an extension of the MGS pattern-matching facilities to handle directly nesting. This makes possible the straightforward emulation of a larger class of unconventional programming models.**

I. Introduction

Modern programming languages allow data structure to be nested so that a valid element of a structure can also be in its turn another structure. Generally, this is not considered in unconventional programming models. For instance, the state of a cell in a cellular automata is not (the state of) another cellular automata. Another example: the value labeling a symbol in parametric Lindenmayer system is not (a string representing a derivation in) another Lindenmayer system.

In *chemical computing*, as exemplified in Gamma [1], chemical solutions are abstracted as *multisets* (a generalization of the notion of set in which members are allowed to appear more than once) and a molecule corresponds to an elementary data and not another chemical solution. Nested multisets are considered in *membrane systems*[1] [4] but are studied as a completely different computational model. Indeed, the management of the nesting entails the introduction of new mechanisms (transport rules in the case of membrane systems).

In this paper we consider arbitrary nesting in MGS, a spatial computing language where space is managed through the structure of the data. MGS relies on neighborhood relationships to represent physical (spatial distribution, localization of the resources) or logical constraints (inherent to the problem to be solved) in a computation.

[1]Nested multisets are also considered in High Order Chemical Language [2]. In Structured Gamma [3], elements of the multiset are linked by relations defined by a graph grammar. It is then theoretically possible to encode a given static nest of multisets using relations specified by a specific graph grammar to implement membership test and to make a distinction between elements and nested multisets.

By considering different classes of neighborhood relationships, MGS can emulate several unconventional computing models from the point of view of the programming. The use of arbitrary nested spatial structures allows a hierarchical form of coupling between them. Furthermore, we propose an extension of the MGS pattern-matching facilities to handle nesting explicitly. This makes possible the concise expression of various algorithms as well as the straightforward emulation of a larger class of unconventional programming models.

Outlines: This paper is organized as follows. The next section introduces the emerging field of *spatial computing* and the notions of topological collection and transformation developed in MGS. Section III discusses the relevance of nesting in the context of spatial computing and section III-D proposes a new pattern matching construct to make the handling of nested structures mores easier. Section IV exemplifies the use of nested spaces with three direct applications. The first encodes terms used to represents boolean formulae with nested sets. The computation of a disjunctive normal form on this representation is explained. The second example computes *quadtrees*, a recursive data structure for partitioning a two dimensional space. The last one is dedicated to the informal translation of the *fraglet* computational model into MGS. Related and future work concludes this article.

II. Background

A. Computing in Space, Space in Computation and Spatial Computing

Spatial Computing is an emerging field of research [5] where the computation is structured in term of *spatial relationships* where only "neighbor" elements may interact.

For example, the elements of a physical computing system are spatially localized and when a locality property holds, only elements that are neighbor in space can interact directly. So the interactions between parts are structured by the spatial relationships of the parts.

Even for non physical system, usually an element does not interact with all other elements in the system. For instance, from a given element in a data structure, only a limited number of other elements can be accessed [6]: in a simply linked list, the elements are accessed linearly (the second after the first,

the third after the second, etc.); from a node in a tree, we can access the father or the sons; in arrays, the accessibility relationships are left implicit and implemented through incrementing or decrementing indices (called "Von Neumann" or "Moore" neighborhoods if one or several changes are allowed).

More generally, if an element e in a system interacts during a computation with a subset $E = \{e_1, \ldots, e_n\}$ of other elements, it also interacts with any subset E' included in E. This closure property induces a topological organization: the set of elements can be organized as an *abstract cellular complex* which is a spatial representation of the interactions in the computation [7]. This abstract space instantiates a neighborhood relationship that represents *physical* (spatial distribution, localization of the resources) or *logical* (inherent to the problem to be solved) constraints.

In addition, space can be an input to computation or a key part of the desired result of the computation, *e.g.* in computational geometry applications, amorphous computing [8], claytronics [9], distributed robotics or programmable matter... to cite a few examples where notions like position and shape are at the core of the application domain.

B. The MGS approach to Spatial Computing

The MGS project recognizes that space is not an issue to abstract away but that computation is performed distributed across space and that space, either physical or logical, serves as a mean, a resource, an input and an output of a computation.

1) Topological Collections: In MGS, the notion of space is handled through a slight generalization of the notion of *field*. In physics, a field assigns a quantity to each point of a spatial domain [10].

MGS handles spatial domains defined by *abstract cellular complex* [11]. An abstract cellular complex is a formal construction that builds a space in a combinatorial way through more simple objects called *topological cells*. Each topological cell abstractly represents a part of the whole space: points are cells with dimension 0, lines are cells with dimension 1, surfaces are 2 dimensional cells, etc. The structure of the whole space, corresponding to the partition into topological cells, is considered through *incidence relationships*, relating a cell and the cells in its boundary.

In this approach a field is a finite labeling of a cellular complex: a cellular complex may count an infinite number of cells but MGS restricts itself on fields labeling only a finite number of these cells. Such fields are called *topological collections* to stress the importance of the neighborhood relationships induced by the incidence relationships. Topological collections are a weakening of the notion of *topological chain* developed in algebraic topology [12] and have been introduced in [13] to describe arbitrary complex spatial structures that appear in biological systems [14] and other dynamical systems with a time varying structure [15], [16]. Topological collections generalize fields because they associate a quantity with 0-cells (points in space) but also with arbitrary n-cells.

Graphs are examples of one dimensional cellular complexes: they are made of only 0- and 1-cells. In this paper, we will stick to topological collections where the underlying complex is a graph. In [6] it has been showed how usual data structures (sets, multisets, lists, trees, arrays...) can be seen as one dimensional topological collections: the elements in a data structure are the quantities assigned by the field to the nodes of a graph.

A specific neighborhood relationship has also a special importance in the rest of this paper: the *full relation*. With this relation, every node is neighbor of every other nodes. This corresponds to a node-labeled complete graph and to the multiset data structure.

2) Transformations: Usually in Physics, fields and their evolution are specified using differential operators. MGS generalizes these operators in a rewriting mechanism, called *transformation*. A transformation is the application of some local rules following some strategy. The application of a local rule $a \Longrightarrow b$ in a collection C:

1) selects a subcollection A that matches the pattern a;
2) computes a new subcollection B as the result of the evaluation of the expression b instantiated with the collection A;
3) and substitutes B for A in C.

A local rule specifies a local evolution of the field: the left hand side (lhs) of the rule typically matches elements in interaction and the right hand side (rhs) computes local updates of the field.

Patterns pat are expressions defined inductively starting from the *pattern variables* "x" (matching exactly one element in a collection) and several operators used to compose more complex patterns: the *neighborhood* between two subcollections "pat, pat'", the *repetition* "$pat\star$", the *guard* "pat/exp" and the *typing* "$pat : T$" (elements matched by pat have to be of type T).

Transformations are a powerful means to define functions on topological collections complying with the underlying spatial structure. For instance, discrete analog of differential operators can be defined using transformations [17]. For multisets, transformations reduce simply to associative-commutative rewriting [18] also called multiset rewriting.

III. Nested Spaces

In classical Physics, a field can be classified as a scalar field or a vector field according to whether the value of the field at each point is a scalar or a vector. However, it is not usual to consider "field valued fields". From the MGS perspective, this notion simply corresponds to the idea of nested topological collections. Such a feature is relevant and valuable in at least three areas: for the representation of and the computation on hierarchical or inductive data structures; in the modeling and the simulation of multiscale systems; and in the emulation of "stratified" computational models.

A. Inductive Data Structures

The possibility to nest arbitrarily data structures is now pervasive in modern programming languages. Its usefulness to represent hierarchical data (*e.g.*, XML) or inductive structure

(*e.g.*, list, trees) is well established. We give an example of the use of nested spaces in an algorithmic application relying on multisets in section IV-A.

B. Multiscale Systems

The modeling of a natural system often implies entities appearing on distinct temporal and spatial scales: each level addresses a phenomenon over a specific window of length and time. These scales appear for logical reasons (at a particular scale, the system exhibits uniform properties and can be modeled by homogeneous rules acting on objects relevant at this scale) or for efficiency reasons (*e.g.*, the reductionist simulation of the whole system from first principles is computationally not tractable while we are only interested in coarse-grained description).

Multiscale models and simulations arise when interactions between scales must be considered. For spatial scales, it means that simultaneous spatial representations must be managed as in *adaptive mesh refinement* [19]. This method relies on a sequence of "nested rectangular grids" on which a PDE is discretized. It is important to realize that these subgrids are not patched into the coarse grid but overlaid to track the feature of interest. A simplistic example is presented in section IV-B. Another example, in the area of discrete modeling, is the *complex automata* framework [20] corresponding to a "graph of cellular automata".

Sometimes scales can be separated, meaning that the coupling between scales can be localized at some isolated interaction points in space and time. Then, the resulting computation corresponds to a hierarchical process with a directed flow of information. This is not always the case and we will introduce a dedicated pattern-matching mechanism in section III-D to ease the reference between scales.

C. Stratified Computational Models

Some models of computation exhibit naturally an inductive structure. For instance, the state of a *membrane system* is a multiset of symbols and (inductively) membrane systems. This structure leads directly to a nested organization of "multiset of symbols and multisets".

Some computational models are also best described as a combination of two paradigms: the second being substituted for some generic parts in the first. We list a few examples issued from various compartmentalization devices introduced over a basic chemical framework. In membrane systems, strings have been considered instead of symbols [21]. This leads obviously to "multiset of sequences of symbols and multisets". Nested multisets are restricted to the description of membranes organized by inclusion only. *Tissue P systems* [22] arrange the membranes and their interactions following an arbitrary graph, calling for a "graph of multisets". *Spatial P systems* [23] are a variant of P systems which embodies the concept of space and position inside a membrane. Membranes and objects are positioned in a two-dimensional discrete space. Hence, we have to consider "grids of multisets and grids and symbols".

In section IV-C, we will sketch the encoding of *fraglets*, a molecular biology inspired execution model for computer communications leading to "graph of multisets of sequences".

D. Matching Nested Structures

In the next section, we give examples of the three usages of nested spaces we have identified above section III. The use of nested spaces does not require *a priori* new control structures. For example, if the reactions between symbols of a P systems are coded by a transformation $EvalRule$, then we can defines a function $Apply$ and an auxiliary transformation $ApplyNested$ to thread $EvalRule$ over the nested structure:

```
fun Apply(x) = EvalRule(ApplyNested(x))
and trans ApplyNested = x:bag ⟹ Apply(x)
```

This piece of code is enough to trigger the chemical rules specified by $EvalRule$ through the entire structure. But the transport rules, used for example to expel one molecule from a membrane to the enclosing one, are a little bit heavier to write because they imply the simultaneous matching of two levels in the nested structure.

The usual MGS pattern constructions are "flat": the pattern variables of a transformation T refer to elements of the collection on which T is applied [24]. To make easier the handling of nested collections, we extend the current syntax with a new construction allowing references to elements of a nested collection. The pattern construct

```
[ pat | x ]
```

matches a collection C nested within the current one. The pattern pat must match a subcollection C' in C and the variable x is bound to the collection C deprived of C'. For example, the pattern

```
x, [2, 3 | y] as z
```

matches only one occurrence in the sequence

```
(0, (1, 2, 3, 4), 5)
```

and binds x to 0, z to the nested sequence `(1, 2, 3, 4)` and y to the sequence `(1, 4)`. The notation `[ pat | ... ]` can be used to spare a variable if the rest of the subcollection is not used elsewhere.

IV. Computing with Nested Collections

A. Disjunctive Normal Form

Since the logical conjunction and disjunction operators are associative, commutative and idempotent, a logical formula can be encoded by nested sets. Let consider the following type declaration:

```
type formula = string | Not | And | Or
and record Not = { f:formula }
and collection And = set[formula]
and collection Or = set[formula]
```

In this declaration, `formula` is a sum type: a formula is either a boolean variable (represented by a string value), or the negation of a formula nested in a record with one field `f`, or the conjunction (resp. disjunction) of formulae nested in a set with subtype `And` (resp. `Or`). With these types, the formula $\neg(p \wedge q) \vee r$ can be represented as follows:

```
{ f = "p"::"q"::And:() }::"r"::Or:()
```

where e::S inserts an element e in the set S and C:() denotes the empty collection of type C.

The computation of the disjunctive normal form can be achieved by iterating until a fixpoint is reached, the following transformation:

```
trans DNF = {
  (* Simplifying unaries *)
  [[x | ...]:Not | ...]:Not ⟹ x
  x:And / size(x)==1 ⟹ choose(x)
  x:Or / size(x)==1 ⟹ choose(x)

  (* Flattening nested ops *)
  [f:And | g]:And ⟹ join(f,g)
  [f:Or | g]:Or ⟹ join(f,g)

  (* De Morgan's laws *)
  [x:Or | ...]:Not ⟹
    fold(::, And:(), map(λf.{ f=f }, x))
  [x:And | ...]:Not ⟹
    fold(::, Or:(), map(λf.{ f=f }, x))

  (* Distributivity *)
  [x:Or | s]:And ⟹ map(λf.f::s, x)

  (* Induction *)
  x:And ⟹ DNF(x)
  x:Or ⟹ DNF(x)
  x:Not ⟹ DNF(x)
}
```

Each rule is a straightforward translation in MGS of a well known transformation of a boolean formula into an equivalent one. In this program, the `map` and `fold` are the usual map and fold functions: `map(`f`,`s`)` applies the function f to each elements of the collection s and returns the collection of results; `fold(`f`,`z`,`s`)` reduces the elements of the collection s using the binary function f and starting from z. The function `join` is used to append two collections and `choose` is used to pick-up one element in a collection. Finally, the expression $\lambda f.f::s$ is a lambda expression that appends its argument (here a formula) to the collection s.

B. A Simple Space Subdivisions Scheme

This example shows the building of a quadtree that partitions a set of points in 2D space. Quadtrees recursively subdivide a rectangular spatial domain into four regions. The subdivision is recursively iterated until there is less than n points in each region (we take $n = 2$ in the following). Figure 1 gives an example where the points are more or less aligned along a curve.

This adaptive mesh is described by the following type definitions:

```
type QuadTree = Grid[QuadTree] | Cloud
and gbf Grid = <n,e; 2e=0, 2n=0>
and collection Cloud = set[Point2D]
and record Point2D = { x:real, y:real }
```

GBF are collections with a regular neighborhood whose topology is specified though a group presentation. Here, the GBF

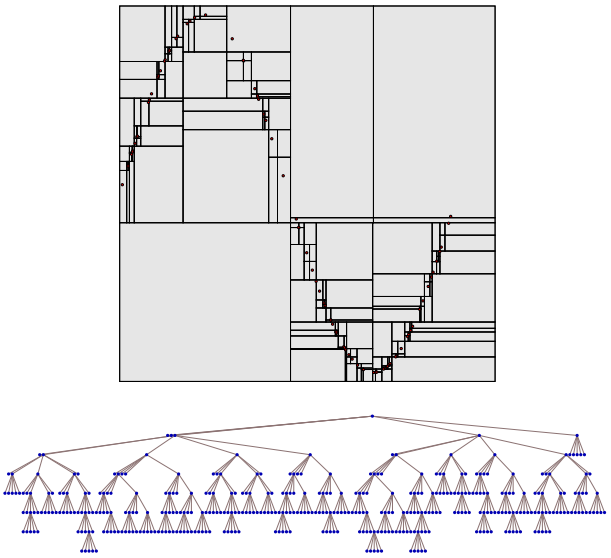


Figure 1. *Top:* Adaptive mesh refinement using quadtrees on a set of 100 points. *Bottom:* The corresponding nested structure pictured as an inclusion tree.

`Grid` specifies a 2×2 torus with two directions `n` and `e`. Such grid describe a partition in four adjacent regions.

The recursive subdivision is computed by the following transformation:

```
trans MakeQuadTree =
  c:Cloud / size(c)>2 ⟹
    MakeQuadTree(SplitCloud(c))
```

The function *SplitCloud* makes the real work:

```
fun SplitCloud(c:Cloud) =
    let g = barycenter(c) in
    let c0, c1 = split(λp.p.x<g.x, c) in
    let c00, c01 = split(λp.p.y<g.y, c0) in
    let c10, c11 = split(λp.p.y<g.y, c1) in
        Grid:(c00@0, c01@e, c10@n, c11@(n+e))
```

Function *SplitCloud* divides a cloud of points c into four subclouds c_{ij} depending on the positions of the points compared to the barycenter (above or below, on the left or on the right). Then it builds a new `Grid` collection where the four cells are labeled by the clouds c_{ij}. Function `barycenter` computes the center of mass of a cloud of points (it is easily expressed using a fold over the elements of the set.) The function `split` takes a predicate and a collection, and returns two collections: the elements of the first one satisfy the predicate and the second one gathers the remaining elements. The syntactic construction T`:(`$\ldots v_i$`@`$c_i \ldots$`)` builds a collection of type T where the value v_i is associated with the cell c_i.

The process is illustrated on figure 1. There is no need of the new pattern matching construct because there is no need to "mix" the elements of a top level collection with the elements of a nested one. It is the recursive calls of *MakeQuadTree* that create the nested structure from a flat cloud of points.

C. Fraglet

Fraglets are tiny computation fragments or sequences of tokens that flow and react through a computer network. They have been introduced in [25] as an execution model for computer communications inspired by molecular biology. They have been designed to lay the ground for automatic network adaption and optimization processes as well as the synthesis and evolution of protocol implementations. Table I sketches the core instructions.

For example, the fraglets below together with a fraglet `[length tail]` located on the same node of the network, will compute the length of `tail` by generating the fraglet `[total n]` (where n is the size of `tail`):

```
[counter 0]
[matchp length empty stop cnt]
[matchp stop match counter total]
[matchp cnt pop cnt1]
[matchp cnt1 split match counter
                  incr counter * length]
[matchp incr exch sum 1]
```

In this program, the fraglets can be interpreted as follows: fraglet `[counter 0]` defines a local variable with initial value 0; fraglets starting with `matchp` define functions; finally fraglet `[length tail]` is the application of function `length` on the list `tail`.

In the following we encode the fraglet formalism by implementing a fraglet interpreter in MGS. For the sake of simplicity, we do not consider here the localization of the fraglets on the nodes of a communication network and the communication rules between nodes[2]. Let consider the following MGS type declaration:

```
type Token = int | 'nul | 'exch | ...
and collection Fraglet = seq[Token]
and collection State = bag[Fraglet]
```

The state of the system is represented by a multiset inhabited by a population of fraglets; fraglets are sequences of tokens (symbols or integers). For each fraglet operator, Table I gives the formal fraglet instruction, its informal semantics and its translation into an MGS transformation rule. For example, the split instruction consists in extracting the subsequence of tokens in the fraglet located between the operator (the first element in the sequence) and the first occurrence of the special token $\star$. This operation is straightforwardly translated in MGS: the pattern matches in a fraglet the operator `'split` (the syntactic construction `@0` checks that the operator is located at the first position in the sequence) followed by a subsequence terminated by the special token `'time`. The subsequence is specified by $(x/x$ `!= 'time)`$\star$ that matches a repetition of elements different from `'time`.

[2]Nevertheless the reader is invited to pay attention that this restriction is done for the sake of the simplicity: the whole formalism can be specified in MGS using an additional level of nesting by considering a graph labeled by multisets of fraglets.

V. Related Work

Topological collections are reminiscent of *Data-fields*, studied *e.g.* by B. Lisper [26]. Data-fields are a generalization of the array data structure where the set of indices is extended to all $\mathbb{Z}^n$ (see also [27]). We have introduced the concept of *group based fields*, or GBF [28], [29], to extend data-fields towards more general regular data structures. Topological collections emphasize data structures as a set of *places* independently of their occupation by values. This approach is also shared by the theory of *species of structures* [30]. Motivated by the development of enumeration techniques for labeled structures, the emphasis is put on the transport of structures along bijections while spatial computing focuses on topological relationships.

Disentangling the elements in a data structure from their organization has several advantages. In [31], B. Jay develops a concept of *shape polymorphism* where a data structure is also a pair (*shape*, *set of data*). The shape describes the organization of the data structure (restricted to tabular organizations) and the set of data describes the content of the data structure. This separation allows the development of *shape-polymorphic functions* and their typing: the shape of the result of a shape-polymorphic function application depends only on the shape of the argument, not of its content. The same line is developed in the field of *polytypic programming* for algebraic data type [32]. MGS transformations are naturally polytypic and extend far beyond arrays and algebraic data type. Polytypism in MGS relies on a generic implementation of pattern matching [24] not on overloading or *ad-hoc* polymorphism.

Transformations are a kind of rewriting that differs in many ways from graph rewriting. Their formalization in [33] is not based on the usual graph morphisms and pushouts like in [34] but is inspired by the approach of J.-C. Raoult [35] where graph rewriting based on a (multi-)set point of view is developed. The proposed model is close to term rewriting modulo associativity and commutativity (where the left hand side of a rule is removed and the right hand side is added). This kind of approach also allows to extend results from term rewriting to topological rewriting (as we did for termination in [36]). Note that the notions of topological collection and topological rewriting are more general than labeled graphs and graph rewriting, and may handle higher dimensional objects, a feature relevant in a lot of application areas [37].

Nested data structure are now widespread in programming languages but are less natural in the context of data bases. The importance of organizing the accesses to the element in a complex structure trough primitive operations related to the type constructor is stressed in [38]. In MGS, accesses rely on pattern matching, and the pattern matching constructs reflect the spatial structure underlying a collection. Nevertheless, structural recursion, advocated in [38], is straightforward as showed by the programs in sections IV-A and IV-B.

VI. Conclusions and Future Work

In this paper, we have presented the advantages of nesting in a spatial model of computation, the MGS experimental language. The language MGS, has been used in the context

Op	*Input* *Output*

`nul`	`[nul` *tail*`]` `[]`
	destroy a fraglet
	`[ 'null@0 |` *tail* `]:Fraglet` $\Longrightarrow$ `Fraglet:()`

`dup`	`[dup t a` *tail*`]` `[t a a` *tail*`]` duplicate a single symbol `[ 'dup@0,` t`,` a `|` *tail* `]:Fraglet` $\Longrightarrow$ $t::a::a::$*tail*
`exch`	`[exch t a b` *tail*`]` `[t b a` *tail*`]` swap two tags `[ 'exch@0,` t`,` a`,` b `|` *tail* `]:Fraglet` $\Longrightarrow$ $t::b::a::$*tail*
`split`	`[split s1 *` `s2]` `[s1] [s2]` break a fraglet into two at the first occurrence of * `[ 'split@0, (`x`/`x `!= 'time)*` `as` s_1`, 'time |` s_2 `]:Fraglet` $\Longrightarrow$ s_1, s_2
`pop`	`[pop h a` *tail*`]` `[h` *tail*`]` pop the "head" element of the list "a, *tail*" `[ 'pop@0,` h`,` a `|` *tail* `]:Fraglet` $\Longrightarrow$ $h::$*tail*
`empty`	`[empty yes no` *tail*`]` `[yes]` *or* `[no` *tail*`]` test for empty *tail* `[ 'empty@0,` y`,` n `|` *tail* `]:Fraglet` $\Longrightarrow$ `if size(`*tail*`) == 0 then` $y::$`Fraglet:() else` $n::$*tail*
`sum`	`[sum t` n_1 n_2 *tail*`]` `[t` $(n_1 + n_2)$ *tail*`]` arithmetic addition `[ 'sum@0,` t`,` n_1`,` n_2 `|` *tail* `]:Fraglet` $\Longrightarrow$ $t::(n_1+n_2)::$*tail*

`match`	`[match a` *tail1*`],[a` *tail2*`]` `[`*tail1* *tail2*`]`
	two fraglets react, their tails are concatenated
	`[ 'match@0,` a `|` t_1 `]:Fraglet,` `[`b`@0 |` t_2 `]:Fraglet` $\Longrightarrow$ `join(`t_1`,`t_2`)`

`matchP`	`[matchP a` *tail1*`],[a` *tail2*`]` `[`*tail1* *tail2*`]`
	idem as match but the rule persists
	`[ 'matchp@0,` a `|` t_1 `]:Fraglet as` f`,` `[`b`@0 |` t_2 `]:Fraglet` $\Longrightarrow$ f`,` `join(`t_1`,`t_2`)`

Table I
SUBSET OF THE FRAGLETS CORE INSTRUCTIONS (FROM [25]) AND THEIR MGS TRANSLATION.

of P systems [13] and in several large modeling projects in systems biology [39], [14], [40].

One interest of the spatial paradigm *à la* MGS is its ability to subsume several computational models in a single uniform formalism, as long as one focuses on programming [41], [42]. We showed the benefits of considering nested spatial computing through three kind of examples: in algorithmic, in simulation of multiscale phenomena and in the emulation of other programming models.

The management of nested collections is achieved through three kinds of devices:

1) collections are first-citizen values and can be used as the values of another collection;
2) a specific pattern construction `[` p `|` $\ldots$ `]` makes possible, within the current pattern, to refer to the elements matched by a pattern p in a nested collection;
3) recursive type declarations generate predicates used to constrain the nesting and to control the pattern matching facilities.

These three features together enable a very concise and readable programming style, as exemplified in section III-D. All the presented examples are actual MGS programs, at the exception of some slight syntactic sugar.

The work presented in this paper may be enriched and extended in several directions. The pattern matching we have presented can be seen as operating at an "horizontal level" on the elements of a collection and at a "vertical level" when descending to match some elements of a nested collection. The constructions dedicated to the horizontal level are very expressive, allowing for example the matching of an unknown number of elements. The handling of the vertical level is actually restricted to the `[` pat `|` $\ldots$ `]` operator. Other construction can be designed, by analogy with the vertical level. For example, an operator to allow references through an unknown number of nesting, in a manner analog to the iteration operator "$\star$", would be interesting to mimic path queries in XML. Note however that the distinction between horizontal and vertical level is questionable. An alternative approach would be to unify the nested collection by looking for the spatial relationships holding in the whole structure, irrespectively of the horizontal or the vertical view. The topology of this "flat whole structure" can be build as the topology of a fiber space over the top collection. The investigation of this framework remains to be done.

ACKNOWLEDGMENTS

The authors would like to thanks H. Klaudel, F. Pommereau, F. Delaplace and J. Cohen for many questions, encouragements and sweet cookies. This research is supported in part by the ANR projects SynBioTIC.

REFERENCES

[1] J. Banâtre, P. Fradet, and D. Le Métayer, "Gamma and the chemical reaction model: Fifteen years after," *Multiset processing: mathematical, computer science, and molecular computing points of view*, vol. 2235, pp. 17–44, 2001.

[2] J. Banâtre, P. Fradet, and Y. Radenac, "Programming self-organizing systems with the higher-order chemical language," *International Journal of Unconventional Computing*, vol. 3, no. 3, p. 161, 2007.

[3] P. Fradet and D. Le Métayer, "Structured gamma," *Science of Computer Programming*, vol. 31, no. 2-3, pp. 263–289, 1998.

[4] G. Paun, "Computing with membranes," *Journal of Computer and System Sciences*, vol. 1, no. 61, pp. 108–143, 2000.

[5] A. De Hon, J.-L. Giavitto, and F. Gruau, Eds., *Computing Media and Languages for Space-Oriented Computation*, ser. Dagsthul Seminar Proceedings, no. 06361. Dagsthul, http://www.dagstuhl.de/en/program/calendar/semhp/?semnr=2006361, 3-8 sptember 2006.

[6] J.-L. Giavitto and O. Michel, "Data structure as topological spaces," in *Proceedings of the 3nd International Conference on Unconventional Models of Computation UMC02*, vol. 2509, Himeji, Japan, Oct. 2002, pp. 137–150, lecture Notes in Computer Science.

[7] J.-L. Giavitto, O. Michel, J. Cohen, and A. Spicher, "Computation in space and space in computation," in *Unconventional Programming Paradigms (UPP'04)*, ser. LNCS, vol. 3566. Le Mont Saint-Michel: Spinger, Sep. 2005, pp. 137–152.

[8] H. Abelson, D. Allen, D. Coore, C. Hanson, G. Homsy, T. F. Knight, R. Nagpal, E. Rauch, G. J. Sussman, and R. Weiss, "Amorphous computing," *CACM: Communications of the ACM*, vol. 43, 2000.

[9] B. Aksak, P. S. Bhat, J. Campbell, M. DeRosa, S. Funiak, P. B. Gibbons, S. C. Goldstein, C. Guestrin, A. Gupta, C. Helfrich, J. F. Hoburg, B. Kirby, J. Kuffner, P. Lee, T. C. Mowry, P. Pillai, R. Ravichandran, B. D. Rister, S. Seshan, M. Sitti, and H. Yu, "Claytronics: highly scalable communications, sensing, and actuation networks," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems, SenSys 2005, San Diego, California, USA, November 2-4, 2005*, J. Redi, H. Balakrishnan, and F. Zhao, Eds. ACM, 2005, p. 299. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098964

[10] G. T. Leavens, "Fields in physics are like curried functions or physics for functional programmers," Iowa State University, Department of Computer Science, Tech. Rep. TR94-06b, May 1994.

[11] A. Tucker, "An abstract approach to manifolds," *The Annals of Mathematics*, vol. 34, no. 2, pp. 191–243, 1933.

[12] J. Munkres, *Elements of Algebraic Topology*. Addison-Wesley, 1984.

[13] J.-L. Giavitto and O. Michel, "The topological structures of membrane computing," *Fundamenta Informaticae*, vol. 49, pp. 107–129, 2002.

[14] ——, "Modeling the topological organization of cellular processes," *BioSystems*, vol. 70, pp. 149–163, 2003.

[15] J.-L. Giavitto, "Topological collections, transformations and their application to the modeling and the simulation of dynamical systems," in *14th International Conference on Rewriting Technics and Applications (RTA'03)*, ser. LNCS, vol. 2706. Valencia: Springer, Jun. 2003, pp. 208–233.

[16] J.-L. Giavitto and A. Spicher, "Topological rewriting and the geometrization of programming," *Physica D*, vol. 237, no. 9, pp. 1302–1314, jully 2008.

[17] ——, "Topological rewriting and the geometrization of programming," *Physica D*, vol. 237, no. 9, pp. 1302–1314, jully 2008.

[18] N. Dershowitz, J. Hsiang, N. Josephson, and D. Plaisted, "Associative-commutative rewriting," in *Proceedings of the Eighth international joint conference on Artificial intelligence-Volume 2*. Morgan Kaufmann Publishers Inc., 1983, pp. 940–944.

[19] M. Berger and J. Oliger, "Adaptive mesh refinement for hyperbolic partial differential equations," *Journal of computational Physics*, vol. 53, no. 3, pp. 484–512, 1984.

[20] A. Hoekstra, E. Lorenz, J. Falcone, and B. Chopard, "Towards a complex automata framework for multi-scale modeling: Formalism and the scale separation map," *Computational Science–ICCS 2007*, pp. 922–930, 2007.

[21] J. Castellanos, G. Paun, and A. Rodriguez-Paton, "Computing with membranes: P systems with worm-objects," in *String Processing and Information Retrieval, 2000. SPIRE 2000. Proceedings. Seventh International Symposium on*. IEEE, 2000, pp. 65–74.

[22] C. Martín-Vide, G. Paun, J. Pazos, and A. Rodríguez-Patón, "Tissue p systems," *Theoretical Computer Science*, vol. 296, no. 2, pp. 295–326, 2003.

[23] R. Barbuti, A. Maggiolo-Schettini, P. Milazzo, G. Pardini, and L. Tesei, "Spatial p systems," *Natural Computing*, vol. 10, no. 1, pp. 3–16, 2011.

[24] J.-L. Giavitto and O. Michel, "Pattern-matching and rewriting rules for group indexed data structures," in *ACM Sigplan Workshop RULE'02*. Pittsburgh: ACM, Oct. 2002, pp. 55–66.

[25] C. Tschudin, "Fraglets-a metabolistic execution model for communication protocols," in *Proc. 2nd Annual Symposium on Autonomous Intelligent Networks and Systems (AINS), Menlo Park, USA*, 2003, pp. 1–6.

[26] B. Lisper, "On the relation between functional and data-parallel programming languages," in *Proc. of the 6th. Int. Conf. on Functional Languages and Computer Architectures*, ACM. ACM Press, Jun. 1993.

[27] J.-L. Giavitto, D. De Vito, and J.-P. Sansonnet, "A data parallel Java client-server architecture for data field computations over $\mathbb{Z}^n$," in *EuroPar'98 Parallel Processing*, ser. LNCS, vol. 1470, Sep. 1998, pp. 742–??

[28] J.-L. Giavitto, O. Michel, and J.-P. Sansonnet, "Group based fields," in *Parallel Symbolic Languages and Systems (International Workshop PSLS'95)*, ser. LNCS, I. Takayasu, R. H. J. Halstead, and C. Queinnec, Eds., vol. 1068. Beaune (France): Springer-Verlag, 2–4 Oct. 1995, pp. 209–215. [Online]. Available: ftp://ftp.lri.fr/LRI/articles/michel/psls95.ps.gz

[29] J.-L. Giavitto, "Rapport scientifique en vue d'obtenir l'habilitation à diriger des recherches," Ph.D. dissertation, Université de Paris-Sud, centre d'Orsay, May 1998. [Online]. Available: http://www.lami.univ-evry.fr/~giavitto/

[30] F. Bergeron, G. Labelle, and P. Leroux, *Combinatorial species and tree-like structures*, ser. Encyclopedia of mathematics and its applications. Cambridge University Press, 1997, vol. 67, isbn 0-521-57323-8.

[31] C. B. Jay, "A semantics for shape," *Science of Computer Programming*, vol. 25, no. 2–3, pp. 251–283, 1995.

[32] J. Jeuring and P. Jansson, "Polytypic programming," *Lecture Notes in Computer Science*, vol. 1129, pp. 68–114, 1996.

[33] A. Spicher, O. Michel, and J.-L. Giavitto, "Declarative mesh subdivision using topological rewriting in mgs," in *Int. Conf. on Graph Transformations (ICGT) 2010*, ser. LNCS, vol. 6372, Sep. 2010, pp. 298–313.

[34] H. Ehrig, M. Pfender, and H. J. Schneider, "Graph grammars: An algebraic approach," in *FOCS: IEEE Symposium on Foundations of Computer Science (FOCS)*, 1973.

[35] J.-C. Raoult and F. Voisin, "Set-theoretic graph rewriting," in *Proceedings of the International Workshop on Graph Transformations in Computer Science*. London, UK: Springer-Verlag, 1994, pp. 312–325. [Online]. Available: http://portal.acm.org/citation.cfm?id=647364.725670

[36] J.-L. Giavitto, O. Michel, and A. Spicher, *Software-Intensive Systems and New Computing Paradigms*, ser. LNCS. Springer, november 2008, vol. 5380, ch. Spatial Organization of the Chemical Paradigm and the Specification of Autonomic Systems, pp. 235–254. [Online]. Available: http://www.springerlink.com/content/g1357n85j8301078/?p=a5c6f79393724a9d88f508d110a8bfe2&pi=6

[37] E. Tonti, "On the mathematical structure of a large class of physicial theories," *Rendiconti della Academia Nazionale dei Lincei*, vol. 52, no. fasc. 1, pp. 48–56, Jan. 1972, scienze fisiche, matematiche et naturali, Serie VIII.

[38] P. Buneman, S. Naqvi, V. Tannen, and L. Wong, "Principles of programming with complex objects and collection types," *Theoretical Computer Science*, vol. 149, no. 1, pp. 3–48, 18 Sep. 1995.

[39] P. Barbier de Reuille, I. Bohn-Courseau, K. Ljung, H. Morin, N. Carraro, C. Godin, and J. Traas, "Computer simulations reveal properties of the cell-cell signaling network at the shoot apex in Arabidopsis," *PNAS*, vol. 103, no. 5, pp. 1627–1632, 2006. [Online]. Available: http://www.pnas.org/cgi/content/abstract/103/5/1627

[40] A. Spicher, O. Michel, and J.-L. Giavitto, *Understanding the Dynamics of Biological Systems: Lessons Learned from Integrative Systems Biology*. Springer Verlag, Feb. 2011, ch. Interaction-Based Simulations for Integrative Spatial Systems Biology.

[41] O. Michel, A. Spicher, and J.-L. Giavitto, "Rule-based programming for integrative biological modeling – application to the modeling of the lambda phage genetic switch," *Natural Computing*, vol. 8, no. 4, pp. 865–889, december 2009, published online: 12 November 2008. [Online]. Available: http://www.lacl.fr/~michel/PUBLIS/2009/naco09.pdf

[42] A. Spicher, O. Michel, and J.-L. Giavitto, *Understanding the dynamics of biological systems*. Springer, 2011, ch. Interaction-based simulations for Integrative Spatial Systems Biology, pp. 195–231. [Online]. Available: http://www.lacl.fr/~michel/PUBLIS/2010/ibss.pdf

Spatial Computing for non-IT Specialists

Steffan Karger, Agostino Di Figlia, Maurice Bos, Andrei Pruteanu, Stefan Dulman
Delft University of Technology, the Netherlands
{s.j.karger, a.difiglia, m.bos-1}@student.tudelft.nl, {a.s.pruteanu, s.o.dulman}@tudelft.nl

ABSTRACT

Designers and architects are showing an increasing interest for intelligent and interactive building environments, employing large numbers of networked embedded devices, often equipped with wireless communication capabilities. Building small prototypes is usually feasible with a central-control approach. As soon as the prototype needs to be scaled up in the commissioned buildings, complexity arises due to the large number of interacting devices.

In this paper, we link the interactive environments applications with the field of spatial computing. As we will show, the two are strongly correlated and spatial computing can prove to be an elegant solution for the problem at hand. Moreover, spatial computing has the potential of uncovering new designs, based on the emergent behavior properties of large-scale networks. We propose a new framework, called IDS (Interactive Design Studio), which allows for exploration of new design possibilities employing networked embedded systems, without the expertise of IT-specialists.

The IDS framework is built on top of the Proto programming language and targets the protoDeck interactive floor. We showcase its capabilities via two application scenarios and confirm its benefits by means of a survey involving architecture students. Finally, we show implementation details of the complete software stack and experimental results from deployment on the embedded platform.

Keywords

spatial computing, distributed systems, interactive design, embedded systems, software framework

1. INTRODUCTION

Recent years have seen an explosion in the number of networked devices embedded into engineered systems. Wireless sensor networks, swarming robots, mobile ad-hoc networks, smart phones and smart appliances are just a few well-known application domains where this has already become a reality. Properties such as flexibility and ease of use make networked systems attractive solutions for problems outside the information technology domain.

Architects show an increasing interest for intelligent and interactive building environments [7]. Current state-of-the-art includes designs such as the *Ada* floor [8]. It is composed of interconnected tiles capable of interacting with the users stepping on it by means of light patterns. Another example is the *Healing pool* by Brian Knep[15], consisting of a projection of organic patterns on the floor; the patterns self-heal after being torn apart by people walking around. These projects emphasize the growing trend of designing complex interactive spaces in private or public buildings [11, 18].

Even though interactivity is achieved with different technologies, a common property is occurring in all applications: computational elements are spread out and fill the design space. They interact with each other and the users in various ways leading to complex behaviors. When surveying the current deployments, we noticed that the current installations usually employ some form of centralized control. For prototypes consisting of a small number of elements, that is not an issue. When scaling up to large setups, centralized control becomes almost impossible and the distributed interaction is simply dropped. When trying to mimic distributed systems, such as in the case of the healing pool application [15], the technology used (projectors, cameras and image recognition, a limited-sized deployment area) implies the need of a specific, carefully controlled environment, considerably limiting the design freedom.

At high level, we believe that the problem architects face when *designing interactive environments* is very close to *the killer application* for the field of *spatial computing*. The correlation between the two topics is obvious and we can break down the application scenario in two parts linked to the bottom-up and top-down design of complex systems:

- With an ever increasing number of computing devices equipped with sensing and actuation capabilities, there is a quest for exploring feasible and interesting interactive designs that make use of embedded platforms. Non-IT specialists need ways to fast prototype ideas on large-scale systems, while abstracting from the underlying technological complexity related to communication protocols, programming languages, operating systems, embedded virtual machines, hardware platforms etc.

- Secondly, the complexity that arises in such distributed systems, in the form of top-down translation of specifications for system behavior into local rules (also called global-to-local compiling) is a challenging research question that has been addressed before for different application domains [17].

Appears in: *Proceedings of the 11th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2012)*, Conitzer, Winikoff, Padgham, and van der Hoek (eds.), June, 4–8, 2012, Valencia, Spain.

Both problems are still open for research in the field of complexity theory in general and spatial computing in particular. To the best of our knowledge, solutions to both problems include human expertise [9]. We do not hold a completely autonomous solution to these problems - we merely attempt to provide a framework in the form of a software tool chain that makes use of distributed computing, sensing and actuation. The framework targets designers and architects - the non-IT specialists - and aims to help them explore various interactive design ideas via spatial computing constructs.

Previous attempts of specifying global system behaviors via spatial computing constructs were targeted at the so-called IT specialists: we refer the reader to a number of spatial computing domain-specific languages (DSLs) made available in recent years, such as Proto [3], Kairos [10] and TOTA [16]. Our framework is built on top of such a programming language, Proto. We further elaborate on this in Section 2.

The framework we present in this paper has been tailored for protoSpace [12] at TU Delft, Faculty of Architecture, Hyperbody Group[6]. The space has an interactive floor, protoDeck, consisting of 189 tiles each equipped with a microcontroller, RGB leds and a pressure sensor (Figure 1). Due to the power requirements of the LED's, the nodes are powered from the grid. ProtoSpace 3.0 [12] also comprises other multimedia devices such as beamers, a complex sound system and various interactive objects. The ambition is to use protoSpace and all its components as an ecosystem capable to create interactive user experiences. To achieve that, we provide the non-IT specialists with a friendly design tool chain. It facilitates the design of interactive spaces for various events such as art exhibitions, dance performances, teaching activities, social events, etc.

The design tool chain (Figure 2) comprises four components: GUI, StateChart Compiler, DeckSim and protoDeck. They correspond to the four stages of the design process. The GUI serves as a graphical specification tool that eases the description of the tiles' behavior. The GUI produces a state chart representation of the behavior and is given as an input to the StateChart Compiler which generates the platform specific code for DeckSim and the protoDeck hardware. The SC Compiler aims to substitute the embedded systems specialists in the design loop.

The paper has the following outline. In Section 2 we discuss related work for both interactive spaces and spatial computing platforms. Section 3 outlines and describes the framework and its components in detail. An example scenario is given in Section 4. We show the experimental results in Section 5. We discuss the results in Section 6. Finally, we conclude in Section 7.

2. RELATED WORK

Interactive environments have become popular in recent years [5] and several aspects achieving interactivity have been explored. Next section will discuss three interactive spaces (*Ada* [8], *Healing Pool* [13] and *Hallway monitoring* [2]) and their main characteristics in terms of interactivity type and adopted techniques.

2.1 Interactive Environments

Delbruck et al. have created a tactile luminous floor, *Ada*, for an interactive autonomous space. The space consists of a floor, projection screens, microphones, ceiling cameras, speakers and theatre lights. The tiles on the floor are equipped with tactile load sensors and RGB lamps. They are networked as a cellular automata using an industrial automation network, Interbus. A centralized approach is used for controlling the floor's behavior. In fact, the tile's local controller delivers the data to a PC which controls the behavior. In contrast, our approach aims to provide a complete distributed approach to achieve user interactions.

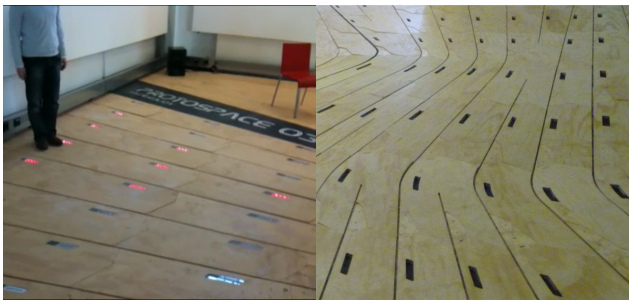


Figure 1: The protoDeck floor. On the left the floor detecting presence of a person, on the right an impression of the shape of the floor.

Another example of an interactive floor is *Healing Pool* [13]. It was presented at the exhibition in the Brauer Museum of Art (Valparaiso University). The *Healing Pool* is an interactive video installation equipped with video projectors, cameras, custom software and a vinyl floor. The main characteristic is the ability to project organic patterns that are torn apart by visitors walking on the floor. Ultimately, they rebuild themselves in an always unique way. Even though the work relies upon artificial intelligence and imaging techniques it shows the strain and increasing interest in interactive spaces. We believe that large-scale complex interaction can be achieved only by means of distributed systems of sensors and actuators via the spatial computing paradigm.

An approach technologically more similar to ours is Hallway Monitoring [2]. In this project, wireless sensor nodes have been placed underneath a hallway floor. The sensor nodes are able to sense pressure on the tiles and actuate lights and speakers on the hallway walls. Due to the limited space and the lack of direct feedback from the tiles, the possibilities for complex interaction are also limited. No extra objects to interact with can be placed in the hallway and the movement of a person is unidirectional only. The set-up offers interesting research possibilities from the computer science viewpoint, but it lacks expressiveness for designers and architects.

2.2 Spatial Computing Platforms

The goal for the protoDeck space is to have an interactive prototyping platform in which architects and designers can develop interactive environments. The spatial and temporal properties are both considered as fundamental constituents. This is strongly correlated to the Spatial Computing paradigm, which endeavors to unleash the potential of using the notions of space and time in programming of distributed systems. In the past, several efforts were taken in this area [4]. In this section we will discuss three of them (Proto [3], Kairos [10] and TOTA [16]). Additionaly, we explain why we chose Proto for this project.

Proto[3] is a functional language that employs the concept of an amorphous medium abstraction[1], in which the discretization of space and time is hidden from the end user. When using Proto, programs do not incorporate their own algorithms for communication and communication related services (e.g, neighborhood discovery or distance estimation). The information about the network and neighborhood is presumed to be available and should be taken care of by the underlying layers. These features enable Proto programs to be very compact. Proto comes with a tool chain that includes a compiler, a simulator and a virtual machine.

Kairos[10] is based on ideas from shared-memory parallel programming. It delivers three primitives: a node abstraction, delivering the programmer tools to manipulate (lists of) nodes, a list of one-hop neighbours and remote data access. Remote data access does not guarantee delivering the correct value, instead Karios relies on 'eventual consistency'. Eventually the system should converge to the correct solution to the problem at hand. While executing tasks, Kairos blocks the execution of the application. Kairos' functionality is delivered through an API, which can be accessed from imperative programming environments. Kairos still remains in a proof-of-concept state.

TOTA[16] stands for 'Tuples over the Air'. It is based on the notion of Tuple fields, which can be seen as information fields from nature, like force fields or chemical gradients. Tuples consist of a content element, a propagation rule and a maintenance rule. Tuples are produced locally and then distributed through the network. Its limitation comes from the fact information from tuples can not be aggregated. TOTA exposes a Java API to the end user.

When choosing a platform we have to remember our goal: an easy to use environment for architects and designers. For this, we need to be able to generate programs from a graphical representation (in our case a state chart) and a tool chain that is feature-complete. The translation from state charts to Proto code is a viable option.

3. SYSTEM DESCRIPTION

The proposed framework is described by the block diagram in Figure 2. The diagram shows four components that correspond to the four stages of our design process. In the following we briefly describe each component and the design rationale behind it.

3.1 GUI

The user interacts with the GUI which consists, in the current state, of a graphical state chart editor. It allows non-IT experts to design a state chart representing the desired behavior of individual tiles. The state chart describes the state transitions of a single tile of the protoDeck floor. The ultimate ambition is to design a user friendly and easy to use GUI for specifying system level and node level behaviors that will hide the cumbersome design of a state chart. The GUI produces an xml file that is structured according to the W3C State Chart extensible Markup Language which serves as an input to the SC Compiler. The rationale behind the use of a state chart representation for the tile's behavior is the following. Since the system under design is reactive and its elements are connected in a mesh topology, the state charts proved to be suitable modeling technique. Moreover, state charts have a way, though limited, of specifying time which is sufficient for our application purposes.

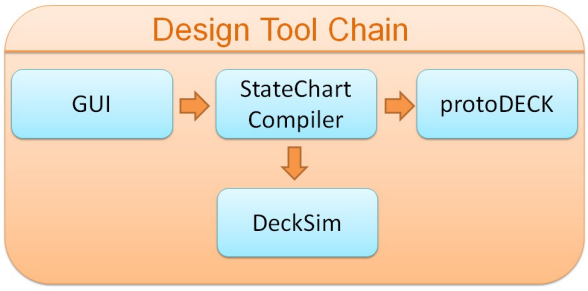


Figure 2: Framework block diagram

3.2 StateChart Compiler

StateChart Compiler is a java based tool that parses the scxml file and produces code for a specific platform or environment. In our case the two supported languages are *Netlogo*[19] and *Proto*[3]. In order to be able to support as many end platforms as possible the specification of the scxml presents a strongly generalized set of events, conditions and actions that can easily be mapped to any specific platform code. In our specific case, the end platforms are DeckSim and protoDeck. The former is a Netlogo based simulation environment, while the latter consists of a mesh network of embedded system devices running the *DelftProtoVM*. The compilation process is a customizable process which receives as input a configuration file describing the used hardware platform in terms of its sensors and actuators and ,in addition, it uses the language specific spatial computing libraries. The SC Compiler interprets the scxml by translating the state chart with the provided hardware specifics. While, the spatial actions are mapped by referencing the provided language specific library. The SC Compiler can be further extended by adding a desired new language specific library and translator module.

3.3 DeckSim

DeckSim is a Netlogo based simulator that provides the possibility to test and have visual feedback of the state chart behavior diagram. The simulation models protoDeck behavior and allows to simulate and test interactions. This way a speedup of the design process can be achieved. The iterative design process consists of a design and test cycle that is usually performed by a designer or architect during sketching or prototyping. They are able to iterate from the specification phase to the test phase and back before deploying the code to protoDeck. Simulations can be run either stand alone or guided.

3.4 Embedded Software Platform

3.4.1 DelftProto VM

The DelftProto VM is a virtual machine that executes Proto bytecode. In September 2011 it replaced the original virtual machine in the Proto distribution, the 'Proto Kernel'. The DelftProto VM code is written to be extremely portable; we were able to succesfully run it on ARM Cortex, Atmel ATmega, MSP430, Intel 586 and AMD 64.

The instruction set of the VM is designed specifically for spatial computing applications. It incorporates instructions that form an aggregate from neighborhood information and common (high level) data types such as vectors are natively

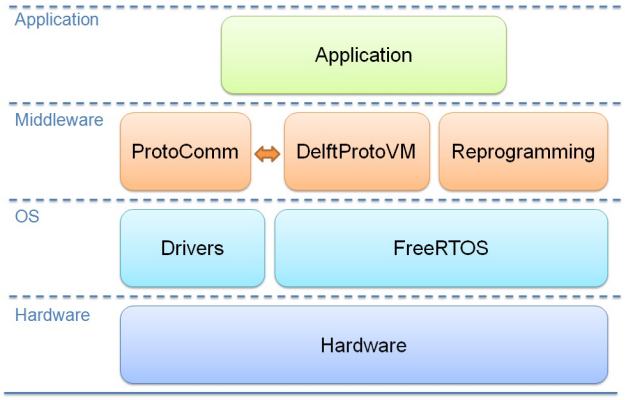


Figure 3: Schematic view of platform components.

supported. Most instructions have implicit operands and work on multiple data types, which allows complex programs to be compiled to very small binaries that can be executed by the VM. For example, a simple gradient algorithm has a size of 35 bytes.

An improved virtual machine based on the DelftProto VM, the *Delft VM*, is currently being developed. Whereas the DelftProto VM is specifically made for programs written in Proto, the Delft VM supports other languages as well. The instruction set makes it easier to generate code from an imperative language, although the focus still lies on functional languages.

3.4.2 Communication and Scheduling

When an application is translated to local rules, it is compiled to run on the virtual machine. We provide an embedded software framework that is easily portable to different hardware platforms. A schematic view of the building blocks is shown in Figure 3.

We use protoDeck as prototyping platform(Figure 1). Each tile is equipped with RGB leds and a pressure sensor. The nodes beneath the tiles are based on NXP LPCXPresso LPC1769 (ARM Cortex-M3) modules connected in a wired mesh configuration. Inter node communication uses the chip's UARTs at 115K2 baud, but our implementation is built to be easily adapted to other communication methods. A wireless (2.4 GHz, 802.15.4 based) version is planned for future experiments. The floor is able to interact with other objects in the room, for example tables, chairs, external lighting and beamers.

The first software layer consists of the FreeRTOS operating system and hardware-specific driver libraries. These deliver basic facilities for the layers on top. The middle layer consists of three parts: the communication library called ProtoComm, the DelftProto VM and a reprogramming facility to update Proto applications virally.

The ProtoComm library supplies the VM with neighborhood information in a best-effort way, since achieving perfect knowledge of all neighbors is generally not possible in real world applications. It takes care of neighborhood discovery, distance estimation, lag estimation, exchange of state information and application updates.

ProtoComm is designed to be compatible with multiple communication types. Incoming data is buffered by device driver interrupt routines. ProtoComm scans the buffers for valid packets and processes them. Packet processing that involves changing the state of the virtual machine is postponed until a virtual machine execution round is completed.

The reprogramming library enables the user to virally roll out new Proto applications without the need to update each node manually. Applications consist of (compact) Proto bytecode, what makes updating the application easier and faster compared to updating a complete platform binary. Nodes keep track of their application version and automatically disseminate new applications as soon as a new version is detected in the neighborhood. An update process is initiated by updating a single node with the new application.

For both disseminating state information and application updates, a negotiation based approach (ADV-REQ-DATA) such as in [14] is employed to avoid broadcast storms and hidden terminal problems. For the case of state updates, which are just exchanged between direct neighbors and thus not propagated, we can reduce completion time by replacing the first advertisement after a detected change in local state with a data packet. The negotiation based technique continues to run in the background to take care of the occasional failed initial communication.

4. INTERACTIVE SCENARIO

As an example scenario we propose using protoSpace to enhance art exhibitions. We imagine the space consisting of several interactive components such as the protoDeck and responsive furniture which engage the visitor and guide him through the various art objects. When visitors walk across the room, the floor leaves a colored trail along the visitor's path. The art objects are placed in showcases which are demarcated by the floor by creating a pulsating light circle around them. Whenever a visitor approaches one of the showcases it triggers an increase of the circle's radius which will surround object and visitor. The light patterns will change triggered by different factors such as the number of people that are close to an art object or the crossing of different visitor trails.

The aforementioned scenario can be composed of several sub-behaviors the space performs in a distributed fashion. Such sub-behaviors are, for example, a distance metric or a desired light pattern. For that reason, we divide the scenario in several sub-behaviors. At the current stage of our project, we performed our experiments focusing on two test applications - a gradient application and firefly synchronization algorithm. By creating a gradient we were able to define a distance metric and show the viability of using spatial primitives. Firefly synchronization is a suitable test to assess the viability of using time primitives while continuously stressing the communication layer.

5. EXPERIMENTAL RESULTS

5.1 User Survey

In order to confirm the benefits of IDS we performed a survey amongst architecture students that have used protoSpace in one of their projects. We were interested in three different topics. Firstly, what kind of network of many 'smart' components they would like an environment to be equipped with (Figure 4a). Secondly, we asked what they would like to improve or upgrade in protoSpace (Figure 4a). Finally, we asked what kind of software tools they would like

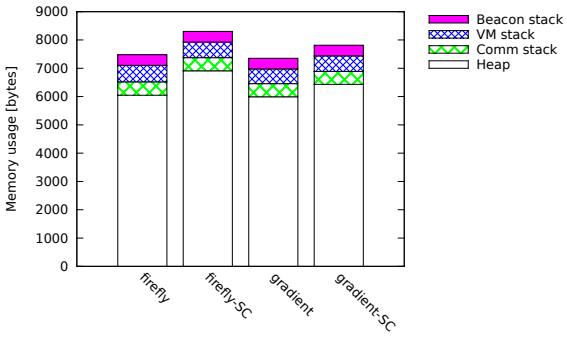


Figure 5: RAM usage comparison for the four Proto applications.

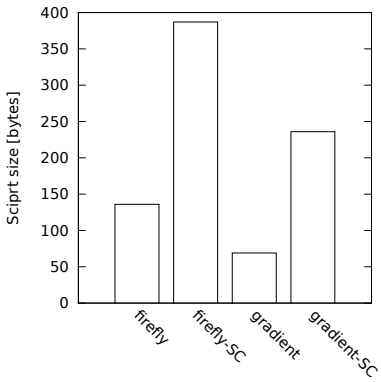


Figure 6: Script size comparison for the four Proto applications.

to have at hand to design or prototype protoDeck or alike (Figure 4c).

In Figure 4 we show the radar charts of the survey outcome. In Figure 4a we can see that the most requested type of 'smart' component is a network of smart furniture followed by a light control system and control software for the space. Smart furniture is for example a chair, table or other object equipped with sensors and actuators. Figure 4b shows the most requested features to add to the existing prototype. These are a designer community, crowd sourcing and more advanced lighting capabilities. Community and crowd sourcing are in a certain way related. With community the students mostly intend a forum like website where they can discuss current and past projects regarding protoSpace. With crowd sourcing we mean code sharing. Finally, Figure 4c shows that most of the surveyed students would like to have a simulation tool or a web based application that is capable to simulate protoSpace. The gathered information is used to guide the ongoing research.

5.2 Simulations versus Testbed Evaluation

As most embedded software developers will acknowledge, the step from simulation to deployment is far from trivial. In this section we highlight several issues we came across while making this step.

Most simulations are based on *unrealistic assumptions*, like instant communication, infinitesimal computation times and a certain level of synchrony. In the actual system usually they do not hold. Our specific approach to handle this is still work in progress.

A larger number of nodes implies a higher risk of *hardware failures*. We conducted some experiments with the previous generation of protoDeck nodes. The testbed consisted of 24 nodes (3 by 8 mesh) with wired serial connections. The connections were not amplified and cables were connected directly to pin headers. This set-up was working when faced with a number of hardware failures. People walking on the floor caused wires to loose contact or suffer from breakage and caused some nodes to die. The high number of components made these failures a rule rather than an exception. This calls for software that can handle failing nodes, unreliable communication.

There are possible *software failures* that will not show until the software is run on the actual embedded platform. However, once on the hardware, it is much more difficult to find the cause of the failures. Getting the simulated environment closer to the real world scenario would help to discover failures earlier.

5.3 Memory Usage and Script Size

To asses *IDS*, we implemented two spatial computing applications, *building a gradient* and *firefly synchronization*. Both are implemented as a manually crafted Proto application as well as an application generated by IDS. Tests are run in a five node network in which one of the nodes was connected to three neighbors. Memory usage statistics were collected from this node.

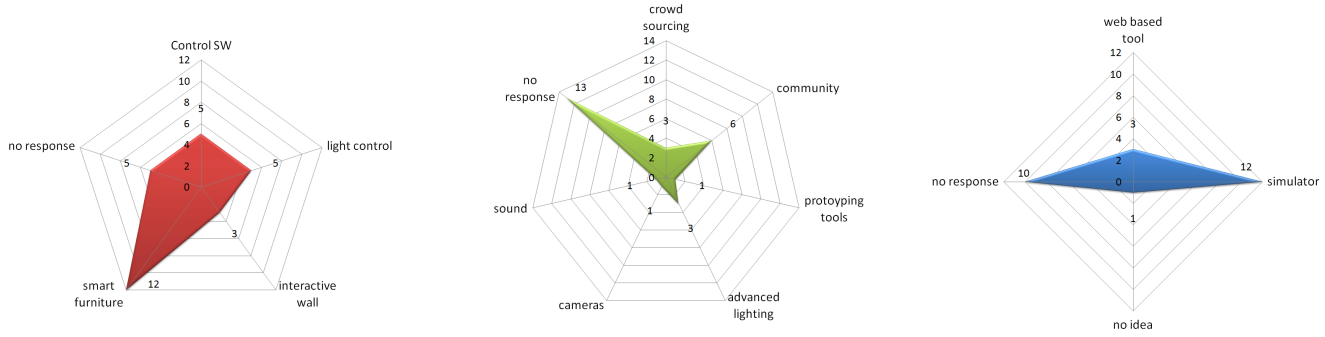
Shown in Figure 5 is the maximum memory usage in bytes for the various applications. As can be observed, the bulk memory is consumed by the heap. The heap is used for storing incoming packets and during execution of the virtual machine. Our primary interest here is the increase in memory usage with increasing application complexity, not the absolute memory usage. When considering script size, shown in Figure 6, as a measure for application complexity we can observe that the increase in memory usage is approximately 10% when the application complexity more than doubles. This indicates there is space for implementing much more advanced applications.

Also note that although the manual and state chart versions of applications are functionally equal, there is a notable difference in script size. This is an indication that there is much to gain from further optimization within the Proto compiler. A compiler should ideally be able to reduce the state chart version of an application to the same script size as the manual version.

6. DISCUSSION

The trend in designing interactive environments is the driving force for creating a toolchain to ease the development of interactive spatial computing applications. To confirm this, we conducted a survey amongst designers and architects. The survey confirms our expectations on the interest in such environments and the need for a simulator to enable fast prototyping. During design toolchain testing, we realized that the current GUI requiring the use a state chart representation is still a cumbersome for designers or architects. Therefore, another level of abstraction is needed to hide the state chart representation.

In an effort to validate the viability of spatial computing



(a) 'Smart' architectural components (b) Requested features (c) Desired tools

Figure 4: Survey results

for interactive environments, we implemented a toolchain that is capable of designing an application using state charts, testing the application in a simulator and running it on actual hardware. A proof of concept implementation of the spatial computing primitives confirms the validity and shows the viability of this approach. For larger applications there might be a need for further optimization of the memory consumption.

7. CONCLUSIONS AND FUTURE WORK

In this paper we introduce a software platform called *IDS* that uses the concepts of *Spatial Computing* to facilitate to non-IT specialists the fast-prototyping of interactive designs using distributed embedded systems installations. It is able to translate high-level specifications into agent behaviors and local interaction rules. We evaluate it via two application scenarios in order to link together all the components of the system. Comparison to related work showed that our approach is one of the first fully-distributed embedded platforms that makes use of the *Spatial Computing* paradigm for fast-prototyping of interactive design installations. As future work, we identified several directions. We will run large-scale experiments by making use of the entire size of the *ProtoDeck* floor. Secondly, the GUI will hide some of the complexity related to expressing agent-level behaviors and will contain more complex aggregate primitives. Complete design ideas will be prototyped and tested in order to improve our methodology based on user feedback.

8. REFERENCES

[1] H. Abelson et al. Amorphous computing. *Communications of the ACM*, 43(5):74–82, 2000.

[2] T. Baumgartner, S. Fekete, T. Kamphans, A. Kröller, and M. Pagel. Hallway monitoring: Distributed data processing with wireless sensor networks. In *REALWSN*. 2010.

[3] J. Beal and J. Bachrach. Infrastructure for engineered emergence on sensor/actuator networks. *Intelligent Systems, IEEE*, 21(2):10 – 19, march-april 2006.

[4] J. Beal, S. Dulman, K. Usbeck, M. Viroli, and N. Correll. *Formal and Practical Aspects of Domain-Specific Languages: Recent Developments*, chapter Organizing the Aggregate: Languages for Spatial Computing. IGI Global, 2012.

[5] T. Bekker, J. Sturm, and B. Eggen. Designing playful interactions for social interaction and physical play. *Personal and Ubiquitous Computing*, 14(5):385–396, 2010.

[6] N. Biloria. Emergent technologies and design. *eCAADe 23*, pages 441–447, 2005.

[7] A. Crabtree, T. Hemmings, and T. Rodden. Pattern-based support for interactive design in domestic settings. In *DIS 2002 Proceedings*, pages 265–276. ACM, 2002.

[8] T. Delbrück, A. M. Whatley, R. Douglas, K. Eng, K. Hepp, and P. F. Verschure. A tactile luminous floor for an interactive autonomous space. *Robotics and Autonomous Systems*, 55(6):433–443, 2007.

[9] S. Dulman. *Robotics in Architecture*, chapter Practical Programming of Large-Scale Adaptive Systems. JapSam Books, 2012.

[10] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, volume 3560 of *LNCS*, pages 466–466. 2005.

[11] M. Haeusler. *Media facades: history, technology, content*. Avedition, 2009.

[12] J. Hubers. Collaborative design in protospace 3.0. *Changing roles; new roles, new challenges*, 2009.

[13] B. Knep. http://www.blep.com/healingPool/.

[14] J. Kulik, W. Heinzelman, and H. Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wirel. Netw.*, 8(2/3):169–185, Mar. 2002.

[15] N. Lehrer and S. Rajko. Thrii. 2010.

[16] M. Mamei and F. Zambonelli. Programming pervasive and mobile computing applications: The tota approach. *ACM Trans. Softw. Eng. Methodol.*, 18:15:1–15:56, '09.

[17] R. Nagpal. *Programmable self-assembly: constructing global shape using biologically-inspired local interactions and origami mathematics*. PhD thesis, Massachusetts Institute of Technology, 2002.

[18] B. Quinn. *Textile Futures: Fashion, Design and Technology*. Berg Pub Ltd, 2010.

[19] U. Wilensky. Netlogo, 1999. http://ccl.northwestern.edu/netlogo/.

Recursivity in Field-Based Programming: the Firing Squad Example

Luidnel Maignan
LIAFA, Université Paris-Diderot
France

Jean-Baptiste Yunès
LIAFA, Université Paris-Diderot
France

***Abstract*—In cellular automata, the well-known firing squad synchronization problems have many solutions usually provided as explicit transition tables, and explained in terms of idealized continuous signals and their collision. However, very few proofs exist despite of the large amount of work on these problems. In this presentation, we take the spatial computing point of view and provide a field-based description of a solution. On the cellular automata part, this provide a understandable and formal construction of a very general solution from which a proof seems to be derivable almost directly. On the spatial computing part, this provides an example of recursive field functional, with a kind of tail-recursivity leading to a strictly finite system.**

I. Introduction

A. Firing squad and signal-based programming

In cellular automata, the firing squad synchronization problem (FSSP) [2], [12], [13] may be stated as follows: *find a finite transition function having a given (fire) state such that, starting from arbitrary sized line of cells where all but one cell (the general) are quiescent, all cells enter for the first time in this state synchronously*. A classical solution is to send signals at different speed so that they first collide at the middles of the space, then at the quarter of the space, then at the eighth of the space, and so on, until an accumulation point is reached. For example, if one sends two bouncing signals from the leftmost cell, one at speed 1 and another at speed $\frac{1}{3}$, these two signals will collide at the middle of the space. However, this is only an idealized presentation since actual solutions have to deal with peculiarities appearing when applying these continuous concepts to the discrete cellular space, the parity of the space being the simplest example. Solutions based on these type on intuition are therefore obtained by solving these peculiarities by hand by iterative correction of the transition table.

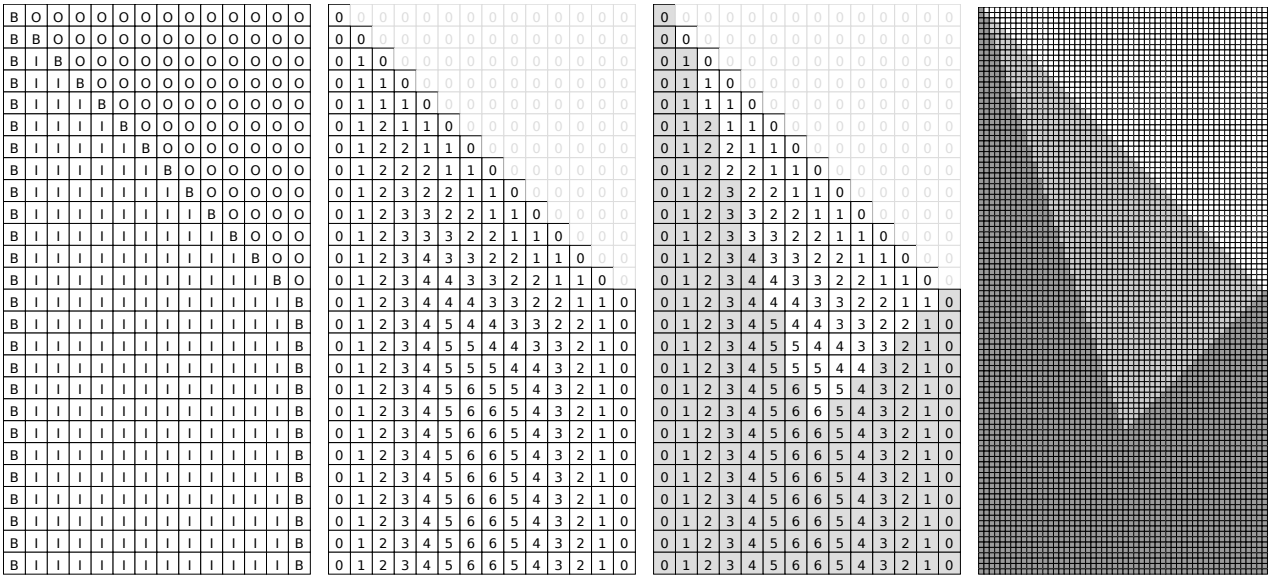
Despite these difficulties, this basic idea has been generalized into many solutions for the classical problems and for some generalizations. One can consider synchronizing with general at any arbitrary position [1], [18], [20], many generals synchronous or not [17], synchronizing 2D-spaces [3], [6], [17], 3D-spaces [16], graphs [5], [15], and variants with different constraints on shape of the space, that may also be dynamic to some extent [4]. However, the drawback of the method is that proofs are hard to obtain directly from the solution, and mistakes has also been found in some cases, using large experiment on many initial configuration. Only a very poor number of proofs of correctness [11], [14], [19] exist.

B. Field-based approaches in spatial computing

Spatial computing considers massively distributed architectures as (programmable) spaces and promote the use of spatial concepts to ease the programming of such architectures. Data structure are therefore spatially extended objects, as can be seen in languages such as PROTO and MGS. In particular, the concept of *fields* is an important one: it is an object which associates a value to each point in space and specifies the local evolution of these values in time. In contrast with cellular automata, the values evolution is not a closed system but an open one, i.e. it may depend on values determined by other input fields. Fields can therefore be composed together to form more complex fields or fully determined (closed) systems, as functions can be composed to obtain more complex functions or programs.

In [7]–[10], the concept of fields has been applied to cellular automata in order to solve algorithmically different dynamic geometric problems in a modular way, using a common set of fields. In particular, the distance field is present in all of them and is the primary building block to collect spatial information in a finite-state manner. While all these problems are geometrical, it was postulated that non-geometrical problems could also be tackled with the same building blocks, and this paper is here to provide such an example. Also, while fields are really manipulated as functions, no cases analogous to recursive functions arose naturally from previously considered problems, this paper is again here to provide such an example, along with a notion of tail-recursivity.

Indeed, we propose to apply the same methodology to provide an algorithmic description of the FSSP. By algorithmic, we mean decomposing the problem into easier sub-problems, solving each sub-problem by a field, and composing the fields together to obtain a cellular automata solving the problem. In this process, each sub-problem and field will have a simple and fully independent semantic, contrary to signals whose meanings depend on all the signals present in the system. Ultimately, we show that we recover the classical concepts of modularity, reusability, semantic decomposition, etc in the context of cellular automata. Benefits of this approach are directly observable by the generality of the provided solution, and by how easier and intelligible a proof for this system seems to be compared to previous solutions.



(a) Region field (b) Distance field (c) Fixity field (d) Size 54

Fig. 1. The three fields describing the initial region and its middle

II. A Field-Based Description of the FSSP

A. An algorithmic decomposition

We recall that the main idea is to divide the space into two equals regions and to proceed recursively until all regions have size 1.

Naturally, the first step is to *identify the middle of the physical space*. To do this we introduce three fields. The first field, R^0, represents the discovery of the region to be cut. The second field, D^0, will provide some distance information deduced from R^0, such that this distance will eventually allows us to detect the middle. The third one is a boolean field, F^0 that indicates the correctness of the values of R^0 and D^0. Indeed, R^0 and D^0 are dynamic fields, which means that R^0 discovers the space from time to time and so its derived field D^0 also updates accordingly. Eventually R^0 stabilizes when it corresponds to the whole physical space and leading in turn to the stabilization of D^0 (see Section II-B).

Now that we have a region and its middle, we introduce another collection of three fields: R^1 that represents the discovery of the two regions induced by the previous cut of the space, D^1 the distance field deduced from R^1 such that the middles of the two regions will be detected, and F^1 the corresponding correctness field. As the reader might guess, this extends to a recursive schema which defines R^ℓ, D^ℓ and F^ℓ in terms of $R^{\ell-1}$, $D^{\ell-1}$ and $F^{\ell-1}$ (see Section II-C).

This obviously implies that we need an unbounded number of fields. However, we will later explain how this can be reduced to a finite system (see Section II-D).

B. Initial region and its middle

The *initial region field* R^0 is defined using three states O ("*outside*"), B ("*border*") and I ("*inside*"). O is the quiescent state of the field. A cell in state O will turn into B as soon as one of its neighbors is in state B. The "border" state is used to mark the border of the region currently discovered, which at this step must finally correspond the whole physical space. A cell in state B which does not coincide with a physical border of the space updates its state to I. With the help of a given static boolean field $Border^0(x)$ that states for each x if it is a physical border or not, the field R^0 is formally defined by:

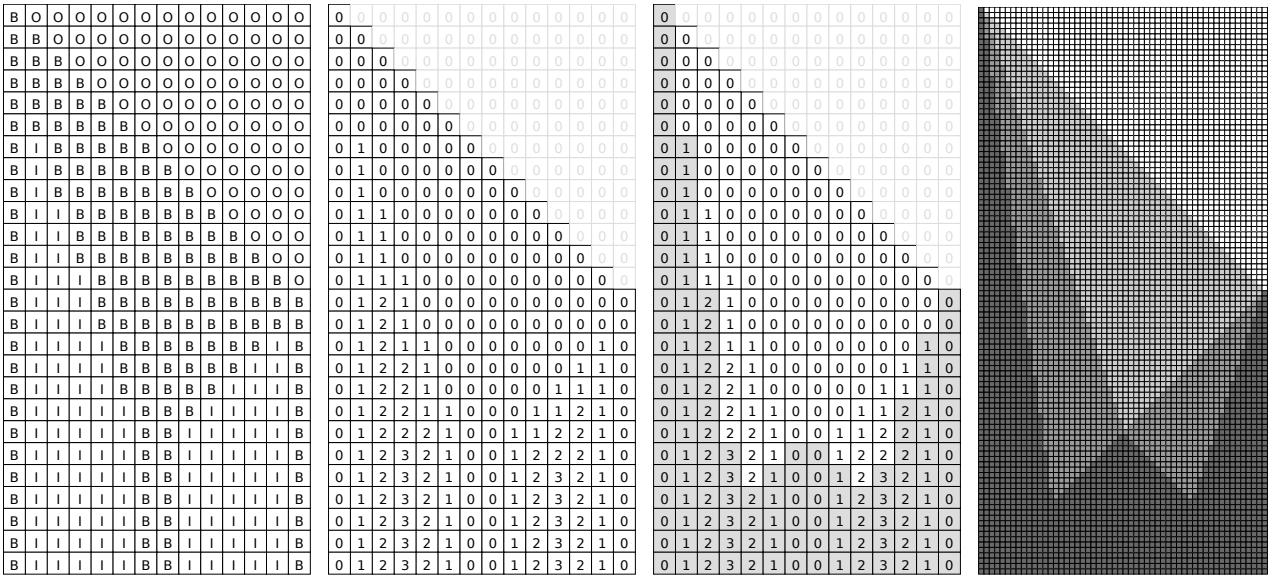
$$R^0_t(x) = \begin{cases} B & \text{if } R^0_{t-1}(x) = O \wedge \\ & \quad \exists y \in N(x); R^0_{t-1}(y) = B \\ I & \text{if } R^0_{t-1}(x) = B \wedge \neg Border^0(x) \\ R^0_{t-1}(x) & \text{otherwise.} \end{cases} \tag{1}$$

From any initial condition of the form $BO\dots O$, the evolution of R^0 produces a space-time diagram like the one depicted in Fig. 1(a).

Now that we have the field that, after some time, represents the whole initial region, we want to determine its middle point. To do so, we use the fact that the middle of a region is the further inner point from both "borders". So, we build the *distance field* D^0, which associates to each cell its distance to latest observed nearest "borders" of the region. As the middle is necessarily an inner cell, the value of D^0 of any cell that is not inside is defined as 0. One can note that this is coherent with the fact that a "border" is obviously at distance 0 from a "border". For an inner cell its distance to the latest observed nearest "border" is obviously 1 plus the smallest distance to the latest observed nearest "border" of its neighbors. This is formally defined by:

$$D^0_t(x) = \begin{cases} 0 & \text{if } R^0_t(x) \neq I \\ \min_{y \in N(x)} 1 + D^0_{t-1}(y) & \text{otherwise} \end{cases} \tag{2}$$

This rule has been extensively studied as a generic building



(a) Region field (b) Distance field (c) Fixity field (d) Size 54, level 0 and 1

Fig. 2. The three fields describing the level 1

block in cellular automata that solve different geometric problems (see [7]–[10]). It is sufficient to detect non-strict local maxima from the distance field to obtain the middle cell(s) of the region. This is illustrated in Fig. 1(b) which shows how region and distance fields evolve.

A final piece is required with respect to synchronization: we need to know when values provided by the region and distance fields are finals. Hence, we define a boolean field F^0 which associates to each cell a boolean indicating if its respective region and distance values are definitely correct. It is possible to determine the appropriate value by a simple case analysis. First, no field's value of an "outside" cell is considered correct (it has not been discovered yet). "Border" cell field's values are correct only if they coincide with a physical border. For "inside" cells, we know by construction that they are really inside so this value is always correct, but we still need to ensure that the distance value is also correct. To determine the distance value correctness, we use the fact that the region only grows, which implies that distance values only increase. And, from the point of view of a cell x, this means that once a neighbor is both correct and minimaly-valued in its neighborhood, it will remain such forever. This ensures that the distance value of x will not evolve anymore since it correspond to this fixed value + 1 as specified in Eq. (2). Altogether, this leads to the following formal definition, whose evolution is illustrated in Fig. 1(c).

$$F_t^0(x) = \bigvee \begin{cases} R_t^0(x) = B \wedge Border^0(x) \\ R_t^0(x) = I \wedge \exists y \in N(x); \\ \quad D_t^0(x) = 1 + D_{t-1}^0(y) \ \wedge \ F_{t-1}^0(y) \end{cases} \tag{3}$$

C. Subsequent regions and middles

Now that the initial region is identified and that enough information has been built to divide it, let us proceed by adding new fields to obtain the division and provide sufficient information to recurse.

First, let us clearly identify what we want to build. From Fig. 1, it should be clear that we are going to build one region starting from the left and another starting from the right. However, we shall prevent ourselves to trust our eyes too much, but try to describe what we want by definition.

Let us come back on what we have done for the initial region and do nearly the same here. Given the predicate $Border^0(x)$, what we built is a region field whose values are, after some time, $R^0(x) = B$ for x's that are physical borders, and $R^0(x) = I$ for x's that are not physical borders and so inner cells.

In the region field R^1, we want to obtain as borders all the "borders" obtained at the previous level and new ones corresponding to the middle(s) cell(s) finally obtained at the previous level. Thus, we consider as borders of the two regions all correct x's such that $R^0(x) = B$, and all x's that correspond to correct non-strict local maxima of D^0. We also want to have $R^1(x) = I$ everywhere x is correct and is neither a "border" nor a maximum among its neighbors in D^0. This naturally leads to the following recursive formal definition of the two predicates $Border$ and $Inside$ for any level $l > 0$:

$$Border_t^{\ell+1}(x) = \bigvee \begin{cases} R_t^\ell(x) = B \wedge F_t^\ell(x) \\ \forall y \in \{x\} \cup N(x); \\ \quad D_{t-1}^\ell(x) \geq D_{t-1}^\ell(y) \ \wedge \ F_{t-1}^\ell(y) \end{cases} \tag{4}$$

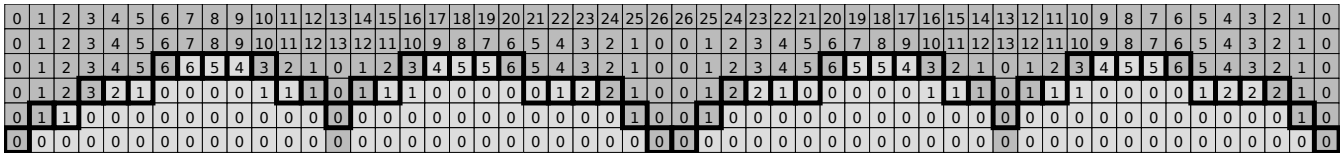


Fig. 3. Stack (level 0 on top) of all field values computed at time 96. Line length is 54.

$$Inside_t^{\ell+1}(x) = F_t^\ell(x) \wedge R_t^\ell(x) \neq B \\ \wedge \exists y \in N(x);\ D_{t-1}^\ell(y) > D_t^\ell(x) \quad (5)$$

Given these two boolean fields, we can apply the same reasoning as before and, obtain nearly the same evolution rule as the initial region. We only need to change the use of $\neg Border^0(x)$ in Eq. 1 into $Inside_t^1(x)$ and the use of $Border^0(x)$ in Eq. 3 into $Border_t^1(x)$. Thus, we obtain the three additional fields describing the first level of division, and iterating this construction, for any level $l > 0$ we obtain the following recursive definition:

$$R_t^\ell(x) = \begin{cases} B & \text{if } R_{t-1}^\ell(x) = O \wedge \\ & \quad \exists y \in N(x); R_{t-1}^\ell(y) = B \\ I & \text{if } R_{t-1}^\ell(x) = B \wedge Inside_t^\ell(x) \\ R_{t-1}^\ell(x) & \text{otherwise.} \end{cases} \quad (6)$$

$$D_t^\ell(x) = \begin{cases} 0 & \text{if } R_t^\ell(x) \neq I \\ \min_{y \in N(x)} 1 + D_{t-1}^\ell(y) & \text{otherwise.} \end{cases} \quad (7)$$

$$F_t^\ell(x) = \bigvee \begin{cases} R_t^\ell(x) = B \wedge Border_t^\ell(x) \\ R_t^\ell(x) = I \wedge \exists y \in N(x); \\ \quad D_t^\ell(x) = 1 + D_{t-1}^\ell(y) \ \wedge \ F_{t-1}^\ell(y) \end{cases} \quad (8)$$

Fig. 2 shows how the three fields evolve at level 1 of the algorithm. In Fig. 2(a) we have one region that grows from the left and starts at the initial time, and another one that grows from the right and starts at time $n-1$ (n is the number of cells). The distance field D^1 evolves inside each region described by R^1.

One can observe that while in D^0 the non-strict local maxima spanned two cells, then in D^1 there is two non-strict local maxima that both span only one cell. This depends on whether the region's length is odd or even (Fig. 2(b)).

D. Reduction to a finite number of states

Now we face two problems. The first one is that distance fields are defined over integers and the other one that we obtained an unbounded number of fields. A detailed explanation of the reducability in finite state is out of the scope of this paper, but let us sketch the most important steps.

The first problem can be solved using a special property. If an integer field is Lipschitz-continuous, i.e. the difference of values between two neighbors is bounded, and the information used in the system only depends on this difference, then it can be transformed into a finite-state field (refer to [8] for all the details). An application of this result is that when the difference is at most 1, then only 3 states are required. With definitions given in Eq. 2 and Eq. 7, it's easy to remark that all the distance fields D^ℓ can therefore be represented with only 3 states each.

To solve the second problem we remark that *in some sense* the recursive schema is "tail-recursive". Indeed, tail-recursiveness is about conserving only the information that are required by the subsequent recursive calls. From the point of view of a cell x, if its field values at given level ℓ are correct, this means that they do not evolve anymore. If furthermore its field values at $\ell + 1$ are also correct and so are the values of its neighbors, then its values at level ℓ are no more useful and can be discarded. This is observable in Fig. 3 where fields values are represented for all cells at a given time. Values (x, ℓ) in darker gray are correct ($F_t^\ell(x)$ is true), and if the whole neighborhood at the next level is also gray, then (x, ℓ) can be "forgotten". By discarding all these gray values (and a little bit more with a much finer analysis), we obtain for each cell a *lowest useful level* represented by a bold surround in the figure. In fact, these are the only necessary values that need to be stored, along with their associated lowest level number (which can be represented with only three state thanks to the Lipschitz-continuous argument). Altogether, this shows that field values are uniformly bounded, and that only a finite number of fields is required. This imply that we finally describe the behavior of a cellular automaton.

III. Conclusion

Without any modification, the system described in this paper is much more general than one can think. Indeed, in all our description we never use the property that there is only one general on the left. Thus we can naturally expect that it is agnostic to such particularities, and this is exactly the case as one can observe in Fig. 4. We also never assumed that the wake-up of the cells happens one after the other from the general, so that removing the corresponding sub-system, one obtain a solution for arbitrary initial desynchronized configuration.

It seems also possible to compose the same fields in slightly different ways to obtain different kind of solutions or to extend this solution to higher dimensions. We can also expect that a proof of correctness of the solution for all sizes and all initial desynchronized configurations to be much more easier than for classical solutions, each field is simple and almost correct by construction, and so is their composition.

References

[1] Karel Culik. Variations of the firing squad problem and applications. *Information Processing Letters*, 30:153–157, 1989.

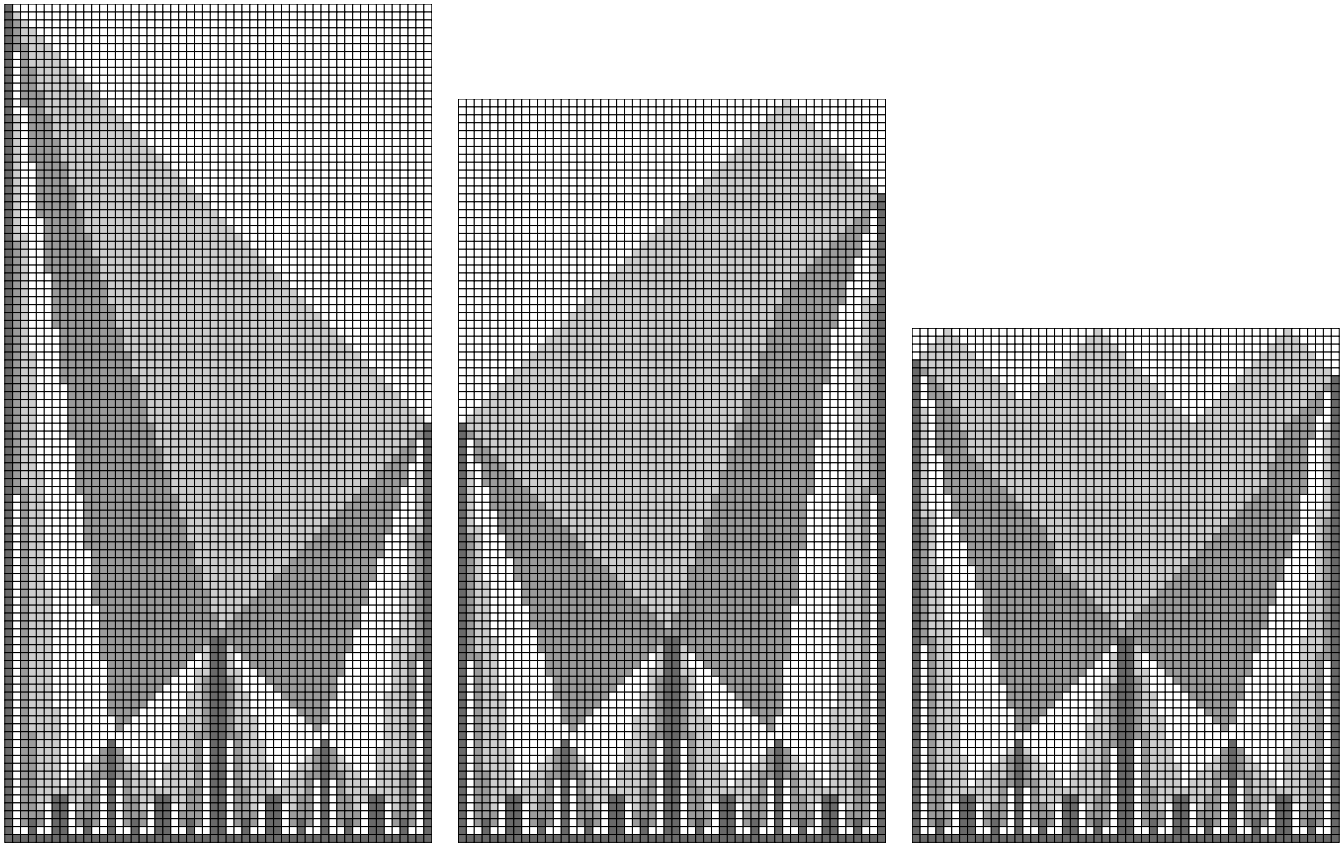


Fig. 4. Evolution of the complete system with different set of generals

[2] Eiichi Goto. A minimum time solution of the firing squad synchronization problem. Courses Notes for Applied Mathematics 298, Harvard University, 1962.

[3] Antonio Grasselli. Synchronization of cellular arrays: The firing squad problem in two dimensions. *Information and Control*, 28:113–124, 1975.

[4] G.T. Herman, W. Liu, S. Rowland, and A. Walker. Synchronization of growing cellular automata. *Information and Control*, 25:103–122, 1974.

[5] Tao Jiang. The synchronization of nonuniform networks of finite automata. *Information and Control*, 97:234–261, 1992.

[6] Kojiro Kobayashi. The firing squad synchronisation problem for two-dimensional arrays. *Information and Control*, 34:177–197, 1977.

[7] Luidnel Maignan and Frédéric Gruau. Integer gradient for cellular automata: Principle and examples. In *Proceedings of the 2008 Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems Workshops*, pages 321–325, Washington, DC, USA, 2008. IEEE Computer Society.

[8] Luidnel Maignan and Frédéric Gruau. A 1D cellular automaton that moves particles until regular spatial placement. *Parallel Processing Letters*, 19(2):315–331, June 2009.

[9] Luidnel Maignan and Frédéric Gruau. Convex hulls on cellular automata. In Stefania Bandini, Sara Manzoni, Hiroshi Umeo, and Giuseppe Vizzari, editors, *ACRI*, volume 6350 of *Lecture Notes in Computer Science*, pages 69–78. Springer, 2010.

[10] Luidnel Maignan and Frédéric Gruau. Gabriel graphs in arbitrary metric space and their cellular automaton for many grids. *ACM Trans. Auton. Adapt. Syst.*, 6:12:1–12:14, June 2011.

[11] Jacques Mazoyer. A six states minimal time solution to the firing squad synchronization problem. *Theoretical Computer Science*, 50:183–238, 1987.

[12] Marvin Minsky. *Computation : Finite and Infinite Machines*. Prentice-Hall, 1967.

[13] Edward E. Moore. *Sequential machines, Selected papers*. Addison Wesley, 1964.

[14] Kenichiro Noguchi. Simple 8-state minimal time solution to the firing squad synchronization problem. *TCS*, 314:303–334, 2004.

[15] P. Rosenstiehl, J.R. Fiskel, and A. Holliger. *Intelligent Graphs : Networks of Finite Automata capable of Solving Graph Problems*. Graph Theory and Computing (R.C. Read Ed.) Academic Press, 1972.

[16] Ilka Shinahr. Two and three dimensional firing squad synchronization problems. *Information and Control*, 24:163–180, 1974.

[17] Helge Szwerinski. Time-optimal solution of the firing-squad synchronization problem for n-dimensional rectangles with the general at an arbitrary position. *Theoretical Computer Science*, 19:305–320, 1982.

[18] V.I. Varshavsky, V.B. Marakhovsky, and V.A. Peshansky. Synchronization of interacting automata. *Mathematical System Theory*, 4 n.3:212–230, 1969.

[19] Jean-Baptiste Yunès. An intrinsically non minimal-time Minsky-like 6-states solution to the firing squad synchronization problem. *RAIRO ITA/TIA*, 42(1):55–66, 2008.

[20] Jean-Baptiste Yunès. Known CA synchronizers made insensitive to the initial state of the initiator. *JCA*, 4(2):147–158, 2009.

Towards a Robust Spatial Computing Language for Modular Robots (Position Paper)

Ulrik Pagh Schultz
Modular Robotics Lab, University of Southern Denmark
Email: ups@mmmi.sdu.dk

***Abstract*—Self-reconfigurable, modular robots are distributed mechatronic devices that can autonomously change their physical shape. Self-reconfiguration from one shape to another is typically achieved through a specific sequence of actuation operations distributed across the modules of the robot. More generally, control of self-reconfigurable robots requires individual modules to act in specific ways in response to sensor input, and these actions need to be coordinated across the modules of the robot. Robust sequential control and role-based control of individual modules has been experimentally demonstrated using the DynaRole language. DynaRole however only allows simple sequences of distributed operations to be executed, which is suitable for self-reconfiguration sequences but lacks the generality needed to implement more complex behaviors.**

In this position paper we present initial ideas on generalizing the DynaRole language to support a wider range of modular robot control scenarios, while retaining robustness, scalability, and the ability to declaratively address issues pertaining to the spatial composition of the robot.

I. Introduction

Modular robotics is an approach to the design, construction and operation of robotic devices aiming to achieve flexibility and reliability by using a reconfigurable assembly of simple subsystems [1]. Robots built from modular components can potentially overcome the limitations of traditional fixed-morphology systems because they are able to rearrange modules automatically on a need basis, a process known as self-reconfiguration, and are able to replace unserviceable modules without disrupting the system's operations significantly. Programming reconfigurable robots is however complicated by the need to adapt the behavior of each of the individual modules to the overall physical shape of the robot and the difficulty of handling partial hardware failures in a robust manner. These challenges bring to mind the use of spatial programming techniques to provide robust and scalable control coupled with the physical shape of the robot.

Control of self-reconfigurable robots can broadly be divided into centralized and distributed approaches. The distributed approaches are considered superior compared to centralized approaches due to their robustness and inherent parallelism, but are on the other hand often intractable in terms of controller design. Centralized approaches are more tractable, but have limited robustness due to having a single point of failure. In earlier work, we have investigated the distributed execution of a pre-specified self-reconfiguration sequence in a modular robot [2]. A sequence is specified using a simple, centralized scripting language, which either could be the outcome of a planner or be hand-coded. The distributed controller generated from this language allows for parallel self-reconfiguration steps and is highly robust to communication errors and loss of local state due to software failures. Furthermore, the self-reconfiguration sequence can automatically be reversed if desired. The scripting language is based on the DynaRole role-based language for modular robots [3], but the distributed scripting facility is only superficially integrated with the role-based control principle, which prompts the development of an improved language which integrates roles and robust, distributed execution.

This position paper reviews the existing work on control of modular robots in the context of spatial computing, with a focus on language-based approaches. Based on this review we propose a generalization of the DynaRole language, named RoCoRo (for Robust Collaobrative Roles). This language incorporates a state sharing feature heavily inspired by the MIT Proto language [4], a notion of distributed scopes for delimiting a modular robot into distinct ensembles of closely collaborating modules, and a generalized approach to robust distributed execution.

II. Spatial Computing and Modular Robots

The term spatial computing denotes collections of local computational devices distributed through a physical space, in which: (1) the difficulty of moving information between any two devices is strongly dependent on the distance between them, and (2) the "functional goals" of the system are generally defined in terms of the system's spatial structure [5]. Modular robots are obviously spatial computing systems: computation and actuation is local to the individual module, communication is in general module-to-module (global communication such as radio could be used, but would hamper scalability), and the typical modular robot application has to do with controlling the physical spatial structure of the system. Modular robots are an interesting application area for spatial computing techniques: space and time are critical given the robotic nature of the system, numerous variations of concrete hardware is available for experimenting with programming, and specifying a global behavior that is compiled into local and robust control is considered a key issue.

Modular robotics has a significant inspiration from biological systems, as is also the case for spatial computing. The individual module is here seen as a cell which is part of a larger multicellular organism. In *homogeneous* systems the modules are physically identical but will typically differentiate their behavior depending on their physical position in

the structure, whereas in *heterogeneous* systems the modules also have different physical characteristics [6]. Chemical and biological concepts such as gradients, hormones and central pattern generators have been used for robust, scalable control of modular robotic systems, although typically in an ad-hoc fashion with an application-specific implementation in C.

A. Modular robot hardware

There are numerous different kinds of modular robots [1]. From the point of view of spatial computing, we can make an overall categorization into macroscale modules, that must be carefully controlled due to motion constraints, and microscale modules, that are typically controlled in a probabilistic way that ignores most if not all physical constraints (such modules so far only exist in simulation). In this paper, we focus on macroscale modules, and we are concerned with the problem of global-to-local compilation of programs for physical modular robots with significant motion constraints, limited processing capacity, and unreliable neighbor-to-neighbor communcation. Microscale modules would typically be more directly amenable to principles of self-organization and mathemetical modelling in general, whereas macroscale modules face many significant implementation issues that must be resolved before these principles become relevant to consider in practice.

As an example of a macroscale module, consider the ATRON self-reconfigurable modular robot (Figure 1), which is our primary experimental platform. The ATRON is a 3D lattice-type system [7]. Each unit is composed of two hemispheres, which rotate relative to each other, giving the module one degree of freedom. Connection to neighboring modules is performed by using its four actuated male and four passive female connectors, each positioned at 90 degree intervals on each hemisphere. The likewise positioned eight infrared ports are used to communicate among neighboring modules and to sense distance to nearby objects. The ATRON exists in two hardware generations: one with an Atmel ATMega128 micro-controller and 4K of RAM per hemisphere, and one with a 1.2MGate FPGA and 64Mb of RAM per hemisphere, in both cases linked by a serial connection. The first generation ATRON is typical of most modular robotic systems: the processing units are severely constrained in order to keep the system simple, realistic to reduce to small size, and potentially cost-effective by mass production. The second generation ATRON is designed as an experimental platform enabling experiments with standard operating systems and programming languages [8].

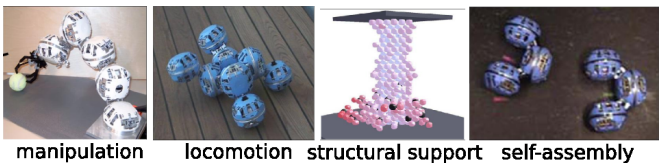


Fig. 1. The ATRON modular robot used for various applications

B. Self-reconfiguration

Self-reconfiguration concerns the spatial transformation of the robot morphology from one shape to another. It is typically viewed as a sequence of operations performed by the robot; in some cases self-reconfiguration could be the only operation performed e.g. if performing locomotion based on self-reconfiguration by shifting the modules towards a specific direction in a caterpillar-like motion.

Off-line planning of self-reconfiguration sequences has been studied for a large number of different robotic systems [6], but is largely complementary to the concerns addressed in this paper: we are interested in providing runtime execution support for control of modular robots, including self-reconfiguration. An off-line planner could use the language proposed in this paper as target, and would thus benefit from its features when performing self-reconfiguration.

On-line, distributed self-reconfiguration algorithms address the execution issue that a number of independent modules must coordinate their actions to perform the correct sequence of actions required for self-reconfiguration [9], [10], [11], [12], [13], [14]. Unlike systems which require an off-line plan to be computed first, these algorithms allow self-reconfiguration to be done automatically given a target shape. However, a limitation of the existing, purely on-line distributed algorithms is that neighbor-to-neighbor communication is essential to determine the position of modules relative to each other, and thus a broken communication link, even if it is only one way, is problematic. For example, in the Proteo system by Yim et al., two-way neighbor to neighbor communication is required for coordination between neighboring modules and propagation of heat values in the heat-based method [14].

For modules with motion constraints, scaling self-reconfiguration to large-scale scenarios is often done by the use of *metamodules:* small, flexible *ensembles* (groups) of closely collaborating modules that can move as a unit through the structure of the robot to shape-change the system [15], [16]. Metamodules emerge from the larger robot configuration, move on the surface of other modules, and stop at a new position. The flow of metamodules, from one place to another on the structure of modules, realizes the desired self-reconfiguration. For the ATRON robot, 3 modules are typically combined into a metamodule; a central module plays the role of a “leg” whereas the two others are attached as “feet”. Programming metamodules has so far been done in a low-level manner using a combination of local actions executed by specific metamodules and global information propagated throughout the structure, such as a gradient serving as attractor for the flow of metamodules [17].

C. Biologically inspired locomotion

Whereas self-reconfiguration typically serves the purpose of transforming the robot between configurations, locomotion is typically performed by actuating modules in a fixed configuration, for example using gait tables [6]. We here review two examples of biologically inspired locomotion that relate both to spatial computing and to the RoCoRo language proposed in

this paper. In both cases, locomotion is achieved by propagating timed communication signals through the spatial structure of the robot.

Shen et al. investigates a hormone-inspired approach to communication and control in the CONRO self-reconfigurable robot, where a set of communication signals triggers different behaviors in modules [18]. Hormone signals are packets that are diffused throughout the structure of the robot, possibly causing operations to activate or new hormones to be created when they arrive. This is similar to chemical diffusion, which has also been used as the basis for spatial computing systems [19], [20], [21], and has been shown to be an effective basis for decentralized communication and execution of programs in spatial computing systems. Shen et al. demonstrate experimentally how hormones can be used to control locomotion and self-reconfiguration of physical modular robots in a highly dynamic fashion that automatically adapts to the current topology. Self-reconfiguration is performed using a "cascade of actions" that in execution is similar to the distributed sequences of DynaRole.

In the work of Stoy et al, a lizard-like structure with four legs is programmed using a primitive form of role-based control where modules respond differently to a time-pulse stimuli that propagates through the structure [22]. The behavior of each module and its response to communication is given by its position in the robot, such as "head", "leg", or "spine". Concretely, roles are used to express how modules interpret sensors and events, and the behavior of each module of the robot at any given time is driven by a combination of its role and timed signals propagated through the structure. In this work, the sole focus is on performing cyclic behavior for locomotion, there is no support for coordination or for performing sequences of actions in response to events.

D. Language-based approaches

The self-reconfiguration and locomotion techniques presented thus far all follow a high-level pattern, but are to the author's knowledge in every case implemented using complicated low-level code that is difficult to reuse in a different scenario. Recently, language-based approaches have however been used in the attempt of creating succinct and reusable software for controlling modular robots.

Locally Distributed Predicates [23], [24] and Meld [25] are two declarative programming languages specifically developed to support the operation of large-scale modular robots composed of spherical microrobots [26] that form a self-reconfigurable spatial computing system. The declarative style of these languages enables complex behaviors of subsets of modules to be derived from concise specifications of spatial constraints. The feasibility of executing these languages on resource-constrainted modular robots has however not been addressed. Moreover, from a language design point of view, the declarative style is perhaps not ideal for specifying complex sequences of operations, as the actual operations to be performed are the result of constraint resolutions as opposed to programmer-specified behavior. This is an open issue that we return to later in this paper. We note that while the context and purpose are similar to the work presented in this paper, a significant difference is the number of modules that robots are anticipated to comprise: The spherical microrobots are assumed to exist in numbers several order of magnitudes higher than macroscale modular robots such as the ATRON. Million-module structures are an ideal match for the typical spatial computing scenario, and can afford to overlook reliability issues that we are intereted in addressing: in the typical macroscale scenario, a single failing module can disrupt locomotion or a whole self-reconfiguration sequence, and must thus be taken into account, while in a highly-redundant context, the same occurrence is often not as significant and can in many cases be ignored due to physical redundancy.

The DynaRole language is designed for role-based control of macroscale modular robots [3]. The DynaRole language is a role-oriented language that allows the programmer to use roles to declaratively specify how behaviors are deployed and activated in the modular robot as a function of its spatial layout and, similarly to the idea of role-based control, how each module responds to sensor inputs and communication. DynaRole programs run on a virtual machine that enables fast and incremental on-line updates of programs, allowing the programmer to interactively experiment with the physical robots. The use of roles allows behaviors to be organized into modules that again are organized into an inheritance hierarchy, providing both reuse and behavioral specialization. Nevertheless, the language provides no support for specifying behaviors at a global level, the underlying virtual machine assumes reliable communication, and in general there is no robustness towards partial failures. Role selection is based on declarative spatial specifications e.g. identifying wheel modules as "modules that have a horizontal rotation axis, only have a single connection, and are at y coordinate 0". The declarative selection primitives and 3D coordinate computations are specific to the robot kinematics, but are in fact automatically generated based on the geometrical description of a single module in the M3L kinematics language [27], which when combined with spatial labels [28] enables morphology-independent programming of modular robots [29].

To enable DynaRole to be used for self-reconfiguration, we extended the language to support robust execution of distributed sequences of operations [2]. Specifically, self-reconfiguration sequences are compiled to a robust and efficient implementation based on a distributed state machine that continuously shares the current execution state between the modules of the robot. Dependencies between operations are explicitly stated to allow independent operations to be performed in parallel while enforcing sequential ordering between actions that are physically dependent on each other. The language is *reversible* meaning that for any self-reconfiguration sequence the reverse one is automatically generated: due to the sequential nature of the programs, any self-reconfiguration process described in the language is reversible by simply performing the corresponding inverse operations in reverse order. Reversibility is subject to physical constraints such as

gravity, changes in the environment, and hardware failures. The continuous diffusion of the state of each module to its neighboring modules provides a high degree of robustness towards partial failures: one-way communication links still serve to propagate state throughout the structure, and modules that are reset (e.g., due to hardware issues or by a watchdog-based timer) are automatically restored from the neighboring modules. Nevertheless, the distributed sequences are extremely simple, there are no conditionals, loops, or propagation of any state except how far the sequence has executed.

III. Analysis

The extension of DynaRole to support execution of distributed sequences provided a significant increase in robustness, which was demonstrated both with (relatively) long-running, reversible self-reconfiguration experiments using physical ATRON modules, and a comprehensive set of self-reconfiguration experiments using simulated ATRON modules (and simulated M-TRAN [9] modules) [2]. One of the primary challenges in programming the ATRON is ensuring robustness towards partial hardware failures in communication, for example two-way communication links that only provide one-way connectivity due to misaligned infrared transceivers. Due to the continuous state diffusion, execution in theory works as long as for any two modules there exists a communication path between them in the robot. The path needs neither to be reliable nor to be static. On the other hand, as mentioned earlier, the sequences cannot react to changes in the environment and are not really integrated with the role-based behavior specification language.

DynaRole sequences could be made more general by adding support for shared program state and conditionals. Shared program state could be diffused similarly to how the distributed sequence progression is shared. In the specific case of sequence progression, each module is reponsible for merging the global state received from neighboring modules with the local state — for arbitrary program state this would have to be handled by the programmer. Such a state sharing approach is inspired by and bears many similarities to MIT Proto [4]: there is not necessarily a single, consistent global state, rather each module continously computes its own view of the shared state. Given that changes to state and progression of execution are propagated together, different parts of the robot may have different views on the state of the sequence, but each of those views will be consistent and will ultimately converge if conditionals are guaranteed to always take the same branch. Indeed, for conditionals the primary challenge is to handle the case where different modules executing parts of the same sequence would take different branches due to sensor inputs or local copies of a shared state having different values. More generally, there is also the question of when to start the execution of a distributed sequence: since the sequence typically involves operations that modify the physical state of the robot, running more than one sequence at a time is usually not relevant. A solution to both of these issues is to delegate the responsibility of triggering sequences and testing conditionals to a single module in the structure. This provides a simple semantics perfectly suitable for e.g. local creation and control of metamodules, but at the obvious cost of limitations in scalability and robustness.

The Meld and LDP language have been designed for controlling subsets of modules within the larger structure. Declarations are used to identify subsets of modules that perform specified operations over time. This approach is obviously required for scalability to larger scenarios, and is essential for supporting the concept of metamodules, which is a proven way of controlling larger-scale ATRON structures. In these scenarios, module groups must be created and dissolved at runtime. A similar scenario is that of self-assembly of modular robots [30]. Here, a larger ensemble is built from smaller groups of modules that become dynamically connected, but the reverse operation splits up the ensemble into smaller groups, each forming their own ensemble. In all these cases, the module subsets can be seen as a dynamic scope delimiter for execution and state propagation. This scope identifies modules that are sharing state and optionally are participating in the execution of a distributed sequence of actions. (This notion of a distributed scope has many similarities to logical neighborhoods [31].)

The design of domain-specific languages often exhibits a tension between declarative and imperative styles of programming. Unlike Meld and LDP which are purely declarative, DynaRole favors a mixed style where declarations are used to control the selection of behaviors in response to the spatial layout of the robot, whereas the behaviors themselves are in an imperative style, similarly the growing point language [19]. We believe the mixed style to be most well-suited to the task of programming modular robots, but this remains an open issue in the design of programming languages for modular robots in particular and for spatial computing in general.

IV. Towards The RoCoRo Language

A. Introducing RoCoRo

We propose the RoCoRo (Robust Collaborative Roles) language as a generalization of the DynaRole language, intended for robust, general-purpose control of modular robots. The language has two primary abstractions: ensembles and roles. An *ensemble* is a dynamic, distributed scope the covers a number of modules and introduces shared state and distributed behaviors into these modules. A *role* applies to a single module, and introduces local state and local behaviors into the module. Roles are further divided into *primary roles* of which only one can be active on a given module at a given point in time, and *mixin roles* of which any number can be active on a given module at a given point in time. A module can be a member of any number of ensembles at a given point in time. Ensembles and roles together are referred to as *entities*, and the set of entities active on a given module is called its *activation*. Declarative rules are used to control the activation of entities based on spatial constraints, the entity activations on neighboring modules, local state from roles, and shared state from ensembles. Entities can be specialized with a semantics

```
abstract ensemble GradiField {
  int g = @MAX_INT;
  g.update {
    min = @MAX_INT;
    for(ng: GradiField.g) min = Math_min(min,ng);
    if(g<@MAX_INT) g = min+1;
  }
}
mixin role GradiSource within GradiField { g.update { g = 0; } }
```

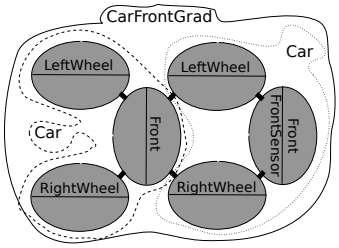


Fig. 2. Gradient field in RoCoRo applied to a configuration of two docked cars using concrete instantiations `CarFrontGrad` and `FrontSensor` which specialize (instantiate) `GradiField` and `GradiSource` respectively. The docked car configuration moreover contains two `Car` ensembles which each consists of three modules playing car-specific roles.

resembling standard object-oriented inheritance: members can be added and existing members can be overridden.

B. RoCoRo by example: gradients

As a classical spatial computing example also relevant in modular robotics, consider the implementation of a simple gradient field in RoCoRo, shown in Figure 2. The ensemble `GradiField` is used to describe the scope of the gradient field, it introduces a shared variable `g` and provides an update rule for `g` which continuously updates the value of `g`. The update rule accesses the available (cached) values of `g` from the neighboring modules, and uses these to compute the local gradient value (the `@` sign indicates an external constant). The mixin role `GradiSource` can be activated on some module that already is part of the `GradiField` ensemble, and overrides the update rule from the ensemble to always make the local gradient value be zero, making the module a source in the gradient field. Member overriding depends on the order at which the entities were activated at runtime, in this case since the role is created within a pre-existing ensemble, the update rule from the ensemble would necessarily be overridden by the update rule from the role. Both entities are however declared abstract, meaning they cannot be activated without first making a specialization for a concrete scenario.

As an example of a specialization of the gradient entities, consider a gradient field for an arbitrary car vehicle; this gradient field is illustrated for a configuration of two docked cars in Figure 2. Assume that the modules in each vehicle are subroles of `Front` or `Wheel`, and that the gradient source should be modules playing the role of `Front` and have no forwards ("north") connections. In this case, the following specialization will activate the gradient field:

```
ensemble CarFrontGrad extends GradiField {
 require subrole(Front) || subrole(Wheel);
}
mixin role FrontSensor extends GradiSource {
 require subrole(Front) && connected(NORTH)==0;
}
```

The ensemble specialization adds a "require" declaration which specifies under what conditions the ensemble can be activated on a given module, and similarly for the mixin role specializing the gradient source.

```
enum Obstacle { None, Left, Right, Center }

ensemble Car {
  // State shared between all modules
  Obstacle obstacle = Obstacle.None;

  // Distributed control behavior
  behavior Front.move() {
    Front.if(Car.obstacle==Obstacle.None) {
      Wheel.drive(); Wheel!evade();
    }
    else {
      Wheel.evade(); Wheel!drive();
    }
  }
}

role Front within Car {
  // Needs 2 neighbors
  require connected(@COMPASS_ANY)==2;
  // Continuously monitor proximity
  behavior checkProximity() {
    if(isProximity(@FRONT_LEFT) &&
      isProximity(@FRONT_RIGHT)) {
      obstacle = Obstacle.Center;
    } else { ... }
  }
}

abstract role Wheel within Car {
  abstract constant Obstacle MY_SIDE;
  abstract constant Compass CONNECTED_SIDE;
  // Require 1 connection + break symmetry
  require connected(CONNECTED_SIDE==1)
       && connected(@COMPASS_ANY)==1);
  // Activated as behaviors by Car.move
  void drive() {
    self.rotateContinuous(100,1);
  }
  void evade() {
    if((obstacle==MY_SIDE)) {
      self.rotateContinuous(50,0);
    } else {
      self.rotateContinuous(100,0);
    }
  }
}

role LeftWheel extends Wheel { ... }
role RigthWheel extends Wheel { ... }
```

Fig. 3. Two-wheeler ATRON car obstacle evasion in RoCoRo

C. RoCoRo by example: obstacle avoidance

One of the primary design goals of RoCoRo is to allow modular robots to be controlled in a robust manner based on a global description of the behavior. As an example of this, consider obstacle avoidance for the small ATRON cars from Figure 1 (rightmost picture), depicted schematically in a docked configuration in Figure 2: a "Front" module in the middle and two "Wheel" modules on the sides. An obstacle evasion program for such a two-wheeler car robot is shown in Figure 3. The ensemble `Car` is used to define the scope of the car; the ensemble is non-abstract and defines no requirements, and so is automatically activated on every module of the robot. This ensemble defines a shared variable `obstacle` (of an enum type, similarly to e.g. C or Java) and a shared behavior `move` which can only be initiated by modules playing the role `Front`.[1] Shared behaviors execute as distributed sequences of operations across the modules of the robot. In this example, the `move` behavior is a global description that expresses the coordination between the various modules of the ensemble; since it only has two steps it could however also have been implemented as a behavior in the role `Front`, but the chosen design arguably makes the overall behavior of the robot more clear. In general, an ensemble behavior can consist of several steps which execute as a robust sequence.

Behaviors are always initiated continuously and atomically to the entity in which they are declared (e.g., for a given role or ensemble, only a single behavior runs at a given point in time). Behavior initiation is decided by a scheduler on the individual module, whether defined on an ensemble or on a role. This is also the case for the behavior `move` which starts with a test on the shared variable `obstacle`. Depending on the value of `obstacle`, the sequence either activates the method `drive` and deactivates the `evade` method or alternatively the inverse, and it does this on all modules of the robot playing the role `Wheel` or a subrole. Activating a method means that it acts like a behavior, that is, it is continuously activated on the respective modules. Deactivating a method means that it stops acting like a behavior. For a given module, this method activation is subject to the state of the distributed sequence being propagated to this module.

The role `Front` defines the requirements for role activation (connected to two modules, the wheels) and its behavior which is to continuously monitor the proximity sensors and update the shared variable `obstacle` correspondingly. Note that no update rules are defined for the shared variable, this means that the default update rule is used, which simply overwrites the local value with the most recent value received from the neighbors, unless the variable was assigned locally in which case it no longer updates automatically but will start to propagate to its neighbors. (If the variable is assigned multiple places in the robot, modules that have not assigned a value to the variable will receive different values at different times through state propagation.) The abstract role `Wheel` defines the conditions under which a wheel is activated as well as how it behaves when the methods `drive` or `evade` are activated. The exact requirement and behavior depends on whether it is a left or a right wheel, which is described by the abstract constants that are defined by the concrete subroles for left and right wheels (not shown). The method `evade` uses a slower rotation speed if the obstacle is on the same side as the wheel, which causes the car to turn away from the obstacle.

[1] If there were multiple modules playing this role, the behavior could be initiated in multiple places at once, dealing with this issue is considered future work.

D. RoCoRo runtime behavior

The RoCoRo language is built on the idea of continuous state propagation by diffusion to neighboring modules. Roles react to changes in the environment that they receive through diffusion, both in terms of what methods are activated and in terms of what role should be active on the module. The shared state from ensembles is also propagated using diffusion, and the local update rules (explicit or implicit) define how the state is merged.

The shared behaviors execute similarly to the distributed sequences from DynaRole: once initiated, each operation in the behavior explicitly denotes the module where it should execute. A program counter denoting the set of parallel executing operations is shared between all modules executing the sequence, and is advanced when modules begin and complete operations. Here, the activation of a method is instantaneous and does not wait for the method to run; rather, the method is activated and will start to run the next time state propagation is performed, and will continue to do so until deactivated.

State propagation is not assumed to be reliable, on the contrary the language is designed for operation of modular robots with unreliable communication links, such as the ATRON. Changes to the module activation, updates to shared variables, activation and deactivation of methods, and progress in the execution of a distributed sequence propagates asynchronously throughout the module structure, and only when the underlying communication system has succeeded in propagating information through a communication link. For consistency, complete information about the state of a module is transmitted in a single packet, but this is problematic on a system like the ATRON where the older generation modules cannot reliably transmit more than roughly 100 bytes of information per packet. We leave this issue to future work.

E. Implementation status

A complete RoCoRo frontend and code generator for Java source code is currently being implemented using the xtext eclipse framework. The generated code assumes a high-level object-oriented runtime system, which is being implemented on top of the USSR generic simulation framework for modular robots [32]. Unlike earlier work [3], this implementation does not address the issue of code distribution in any way, this is considered future work. Moreover, there currently is no underlying spatial information framework, meaning that only very simple predicates can be used to query the physical structure of the robot. Our plan is however to integrate the

M3L language [27] with the simulator to enable automatic generation of new robot implementations from M3L declarations. Such generated robot implementations would be automatically equipped with the ability to compute precise spatial information based on the M3L generation of forward kinematics.

V. Discussion

There are numerous issues that must be resolved before RoCoRo can be used for large-scale scenarios like self-assembly and metamodules. In particular, the semantics of ensembles needs to be defined such that multiple ensembles of the same type can exist in the same robot. We expect that the existing work on logical neighborhoods [31] will be a useful source of inspiration. The RoCoRo language has been pragmatically designed for robust control of modular robots, and we expect that it can be applied as a generally useful programming language for modular robots. The question of how well RoCoRo is suited to other spatial computing tasks, such as programming of sensor networks or swarm robotics, is in an interesting one that will be explored in future work.

Acknowledgment: I would like to thank the anonymous reviewers for their insightful and constructive comments.

References

[1] M. Yim, W.-M. Shen, B. Salemi, D. Rus, M. Moll, H. Lipson, E. Klavins, and G. S. Chirikjian, "Modular Self-Reconfigurable Robot Systems [Grand Challenges of Robotics]," *IEEE Robot. Automat. Mag.*, March 2007.

[2] U. P. Schultz, M. Bordignon, and K. Stoy, "Robust and reversible execution of self-reconfiguration sequences," *Robotica*, vol. 29, pp. 35–57, 2011.

[3] M. Bordignon, K. Stoy, and U. P. Schultz, "A Virtual Machine-based Approach for Fast and Flexible Reprogramming of Modular Robots," in *Proc. IEEE Int. Conf. on Robotics and Automation (ICRA'09)*, Kobe, Japan, May 12-17 2009, pp. 4273–4280.

[4] "MIT proto," retrieved March 8, 2012. Software available at http://proto.bbn.com/.

[5] J. Beal, S. Dulman, J.-L. Giavitto, and A. Spicher, "Spatial computing workshop 2012 call for papers," 2012, http://www.spatial-computing.org/scw12:start, downloaded April 15th 2012.

[6] K. Stoy, D. Brandt, and D. J. Christensen, *An Introduction to Self-Reconfigurable Robots*. MIT Press, 2010.

[7] E. Østergaard, K. Kassow, R. Beck, and H. Lund, "Design of the ATRON lattice-based self-reconfigurable robot," *Autonomous Robots*, vol. 21, no. 2, pp. 165–183, 2006.

[8] M. Moghadam, D. Christensen, D. Brandt, and U. Schultz, "Towards Python-based DSL languages for self-reconfigurable modular robotics research," in *2nd Int. Workshop on Domain-Specific Languages and models for ROBotic systems (DSLRob'11)*, 2011.

[9] E. Yoshida, S. Murata, H. Kurokawa, K. Tomita, and S. Kokaji, "A distributed method for reconfiguration of a three-dimensional homogeneous structure," *Advanced Robotics*, no. 13, pp. 363–379, 1999.

[10] C. Ünsal, H. Kiliccöte, and P. K. Khosla, "A modular self-reconfigurable bipartite robotic system: Implementation and motion planning," *Autonomous Robots*, no. 10, pp. 23–40, 2001.

[11] Z. Butler and D. Rus, "Distributed planning and control for modular robots with unit-compressible modules," *The International Journal of Robotics Research*, no. 22, pp. 699–715, 2003.

[12] M. D. Rosa, S. Goldstein, P. Lee, J. Campbell, and P. Pillai, "Scalable shape sculpting via hole motion: Motion planning in lattice-constrained modular robots," in *Proc. of the 2006 IEEE Int. Conf. on Robotics and Automation (ICRA'06)*, 2006.

[13] S. Murata, H. Kurokawa, and S. Kokaji, "Self-assembling machine," in *Proc. of the 1994 IEEE Int. Conf. on Robotics and Automation*, 1994, pp. 441–448.

[14] M. Yim, Y. Zhang, J. Lamping, and E. Mao, "Distributed control for 3d metamorphosis," *Auton. Robots*, vol. 10, no. 1, pp. 41–56, 2001.

[15] K. C. Prevas, C. Unsal, M. O. Efe, and P. K. Khosla, "A hierarchical motion planning strategy for a uniform self-reconfigurable modular robotic system," in *Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA)*, May 2002.

[16] D. Christensen and K. Stoy, "Selecting a meta-module to shape-change the ATRON self-reconfigurable robot," in *Proc. of IEEE Int. Conf. on Robotics and Automations (ICRA)*, Orlando, USA, May 2006, pp. 2532–2538.

[17] D. J. Christensen, "Experiments on fault-tolerant self-reconfiguration and emergent self-repair," in *Proc. of Symposium on Artificial Life part of the IEEE Symposium Series on Computational Intelligence*, Honolulu, Hawaii, Apr. 2007.

[18] W.-M. Shen, B. Salemi, and P. Will, "Hormone-inspired adaptive communication and distributed control for conro self-reconfigurable robots," *IEEE Transactions on Robotics and Automation*, vol. 18, pp. 700–712, 2002.

[19] D. Coore, "Botanical computing: A developmental approach to generating interconnect topologies on an amorphous computer," Ph.D. dissertation, MIT, 1999.

[20] R. Nagpal, "Programmable self-assembly: Constructing global shape using biologically-inspired local interactions and origami mathematics," Ph.D. dissertation, MIT, 2001.

[21] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli, "Spatial coordination of pervasive services through chemical-inspired tuple spaces," *ACM Trans. Auton. Adapt. Syst.*, vol. 6, no. 2, pp. 14:1–14:24, June 2011. [Online]. Available: http://doi.acm.org/10.1145/1968513.1968517

[22] K. Stoy, W.-M. Shen, and P. Will, "Implementing configuration dependent gaits in a self-reconfigurable robot," in *Proc. of the 2003 IEEE Int. Conf. on Robotics and Automation (ICRA'03)*, Tai-Pei, Taiwan, Sept. 2003, pp. 3828–3833.

[23] M. De Rosa, S. C. Goldstein, P. Lee, J. D. Campbell, and P. Pillai, "Programming Modular Robots with Locally Distributed Predicates," in *Proceedings of the 2008 IEEE International Conference on Robotics and Automation (ICRA'08)*, Pasadena, CA, USA, May 19-23 2008, pp. 3156–3162.

[24] M. De Rosa, S. C. Goldstein, P. Lee, J. Campbell, and P. S. Pillai, "Detecting locally distributed predicates," *ACM Trans. Auton. Adapt. Syst.*, vol. 6, no. 2, pp. 13:1–13:14, June 2011. [Online]. Available: http://doi.acm.org/10.1145/1968513.1968516

[25] M. P. Ashley-Rollman, S. C. Goldstein, P. Lee, T. C. Mowry, and P. Pillai, "Meld: A Declarative Approach to Programming Ensembles," in *Proceedings of the 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'07)*, San Diego, CA, USA, October 29 - November 2 2007, pp. 2794–2800.

[26] S. C. Goldstein, J. D. Campbell, and T. C. Mowry, "Programmable Matter," *IEEE Computer*, vol. 38, no. 6, pp. 99–101, June 2005.

[27] M. Bordignon, U. P. Schultz, and K. Stoy, "Model-based Kinematics Generation for Modular Mechatronic Toolkits," in *Proc. 9th ACM SIGPLAN/SIGSOFT Int. Conf. on Generative Programming and Component Engineering (GPCE'10)*, Eindhoven, The Netherlands, October 10-13 2010.

[28] U. P. Schultz, M. Bordignon, D. J. Christensen, and K. Stoy, "Spatial Computing with Labels," in *Proc. SASO'08 Spatial Computing Workshop (SCW'08)*, Venice, Italy, October 20 2008.

[29] M. Bordignon, K. Stoy, and U. Schultz, "Generalized programming of modular robots through kinematic configurations," in *Proc. of the 2011 IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS)*, 2011, pp. 3659–3666.

[30] K. Stoy, D. J. Christensen, D. Brandt, M. Bordignon, and U. P. Schultz, "Exploit morphology to simplify docking of self-reconfigurable robots," in *Proc. Int. Symp. on Distributed Autonomous Robotic Systems (DARS'08)*, Tsukuba, Japan, 2008, pp. 441–452.

[31] L. Mottola and G. Picco, "Logical neighborhoods: A programming abstraction for wireless sensor networks," in *Distributed Computing in Sensor Systems*, 2006, pp. 150–168.

[32] D. J. Christensen, D. Brandt, K. Stoy, and U. P. Schultz, "A Unified Simulator for Self-Reconfigurable Robots," in *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems (IROS'08)*, France, 2008, pp. 870–876.

On the Space-time Situation of Pervasive Service Ecosystems

Mirko Viroli
Alma Mater Studiorum – Università di Bologna, Italy
Email: mirko.viroli@unibo.it

Graeme Stevenson
University of St Andrews, UK
Email: graeme.stevenson@st-andrews.ac.uk

***Abstract*—We focus on a coordination model for pervasive computing applications, tightly coupled to Semantic Web technologies to support openness and semantic reasoning. Our approach is based on the ideas of reifying the existence of services, data, and events in terms of RDF-oriented annotations maintained by local agents, and enacting global system behaviour by manipulation rules for such annotations, which resemble chemical reactions and can be seen as sequences of SPARQL queries and SPARUL updates over RDF stores. We show a minimal set of ingredients to equip this framework with space-time computing mechanisms, including reification of space-time information in terms of annotations, and a relocation service for annotations. Some examples of spatial computations in pervasive computing are given to illustrate the approach.**

I. Introduction

A *pervasive service ecosystem* is a pervasive computing system in which the various individuals that populate it – such as humans, their smartphones, software services, pervasive displays, sensors and devices spread in the environment, sources of knowledge and data – interoperate opportunistically to achieve their private goals, but are also globally governed by some infrastructure rules analogous to the "laws of nature" in natural ecosystems [22]. Following the general approach proposed in [20], [2], which we here take as a reference, we assume that the presence and activities of such individuals are continuously reflected (in the infrastructure node in which they reside) as semantic annotations, called Live Semantic Annotations (LSA). In concert, these form a global network of annotations representing the virtual counterpart of the physical ecosystem. Overall system behaviour is driven by a set of laws, called *eco-laws*, which act locally to each node, combining and manipulating annotations in a semantic way—enacting all the fine- and coarse-grained processes of LSA interaction, composition, disposal, and so on.

As a concrete implementation of this model, we focus on one that is fully-grounded on standard frameworks and technologies for the Semantic Web, due to their support for openness (supporting interactions with third party software and data) and semantic reasoning (relying on ontologies and semantic matching) [21]. We use RDF as language for structuring LSAs, and the SPARQL/SPARUL query languages for coding eco-laws: the main advantage of this choice is that off the shelf query engines (supporting execution of SPARQL queries and updates over RDF stores) and reasoners [17] can be used to support scheduling and execution of eco-laws locally.

As a key contribution of this paper, which builds on top of [21], [20], we isolate a minimal set of additional middleware services that add the ability to define spatial computing activities (evolutions of distributed structures of LSAs), namely: *(i)* automatically reifying location information in each LSA, *(ii)* reifying a node's spatiotemporal state into new LSAs injected therein, and *(iii)* asynchronously relocating LSAs that have a mismatching location property. A main advantage of the proposed technique is that spatial computations can be structured in terms of chemical-resembling reactions applying semantically to LSAs, handling temporal and spatial aspects in a fully declarative way, treating spacial and temporal aspects no differently from other service properties. We believe this idea can bring new insights as to how spatial features can be added to existing middlewares, particular those which are tuple-based. As a further contribution, we discuss several examples of spatial computations useful in the context of pervasive computing scenarios.

The remainder of this paper is organised as follows: Section II sketches our pervasive ecosystem framework, Section III details the model and its RDF/SPARQL/SPARUL serialisation, Section IV introduces our support for spatial computing aspects, Section V presents example applications, and Section VI provides concluding remarks.

II. Abstract architecture

Pervasive ecosystems [22] are characterised by two main features, which influence their underlying abstract architecture and model. On one hand, they should be *situated*, namely, the activity of any software agent and the data it produces are tightly bound to the agent's physical location: this is because any behaviour should be intrinsically aware of and affect the surrounding context. Situatedness is achieved by infrastructures reifying data, knowledge, and events in the precise point (or region) of space where they pertain, and by promoting interactions based on proximity. Accordingly, a cornerstone of pervasive ecosystems is that a *uniform representation* is required for the various software agents living within them (whether they run on smartphones, sensors, actuators, displays, or any other computational device). We term such a representation a "Live Semantic Annotation" (LSA) for it should continuously represent the state of its associated component (live), and it should be implicitly or explicitly connected to the domain in which such information is produced, interpreted and manipulated (semantic). The LSAs of each agent are reified in a distributed space (called an "LSA-space") acting as the

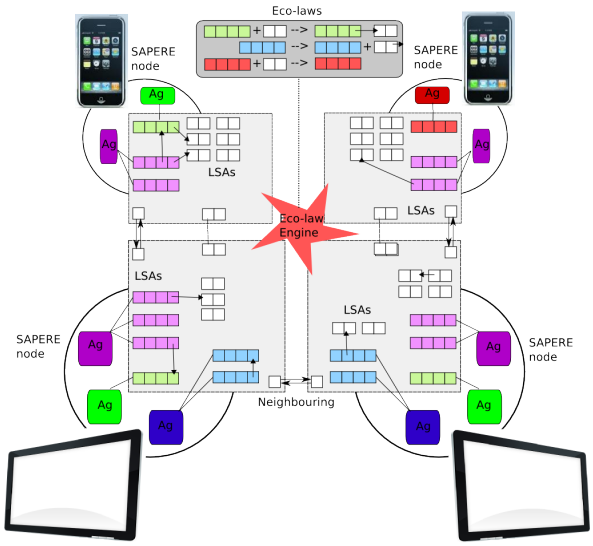


Fig. 1. An architectural view of a pervasive ecosystem.

fabric of the ecosystem, located in the computational device hosting the agent.

On the other hand, pervasive ecosystems should be *adaptive*, exhibiting properties of autonomous adaptation and management to survive contingencies without human intervention and/or global supervision. This is achieved following the natural inspiration [22], by designing system rules that – by acting locally – make global properties emerge dynamically. So, while agents enact their *individual* behaviour by observing their context and updating their LSAs, *global* behaviour (i.e., global system coordination) is enacted by *self-organising* manipulation rules of the LSA-space, which we call *eco-laws*. They can execute delete/update/create actions applied to a small set of LSAs *within the same locality*. Following [3], [18], such eco-laws are structured as chemical-resembling reactions over LSAs.

Figure 1 shows an architectural view, based on the above abstractions, of a portion of an ecosystem featuring: two public displays and two smartphones (carried by people in front of displays), forming a network of 4 computational nodes, each with a local LSA-space containing some running agents (e.g., profile agents and sensor agents in smartphones); LSAs through which agents manifest (in colour); additional LSAs representing data, knowledge, and contextual information like the existence of neighbouring nodes (in white); references from one LSA to another (also called *bonds*); and a set of eco-laws executed by an underlying engine working over the global LSA-space. More generally, one should think of a very large and mobile set of devices connected to each other based on proximity, creating a distributed "space" – ideally a pervasive continuum – where LSAs form spatial structures that evolve over time. The eco-law engine, accordingly, has to be seen as a distributed one uniformly working on all LSA-space—though we will show that a feasible approach amounts at developing local eco-law engines in each node.

III. A CONCRETE MODEL OF LSAS AND ECO-LAWS

A. Live Semantic Annotations

LSAs have a unique, system-wide identifier (LSA-id), and a content (description) including all the information the agent wants to manifest to the ecosystem. This is realised as an RDF-like (Resource Description Framework [10]) set of multi-valued properties, or equivalently, a set of triples that consist of a subject (an LSA-id), a predicate (the property name, a Uniform Resource Identifier – URI) and an object (the assigned value, a literal, URI, or bnode[1]). URIs are qualified by universally-accessible namespaces (using syntax `namespace:term`). In RDF, a literal can be qualified by an XML Schema datatype (XSD) as in `"10.0"^^xsd:double` to enforce type-checking, but we shall omit it for the sake of simplicity, and simply consider quoted strings. By adopting a notation resembling N3 [6], an LSA is represented e.g,. as "`id p v; id q w1 w2 w3;`" where `id` is the LSA-id, property `p` is assigned to value `v`, and property `q` is assigned to values `w1`, `w2`, and `w3`. Following N3, we can avoid repeating the subject when this is unchanged with respect to previous triple, hence writing "`id p v; q w1 w2 w3;`" for the above example. A concrete example of an LSA is, hence:

```
lsa:crowdsensorlsa1123
    eco:type msm:crowd;
    msm:time "2011-05-30T11:00:00";
    msm:crowd_level "0.9";
```

which is the LSA injected by a sensor that describes the precise point in time that a value (0.9) concerning the presence of people in a given room of a museum is sensed.

B. Eco-laws

Eco-laws are structured as chemical-resembling rules of the kind "`P+..+P --r--> Q+..+Q`". Elements `P` and `Q` are patterns of LSAs, expressed like LSAs with the following extensions: *(i)* in place of each element of a triple one can use a variable `?V` (matching any value) or an annotated variable `?V(filter)` where *`filter`* is a predicate expression over `?V` (matching any value that makes *`filter`* true); and *(ii)* the object of a triple can be prepended by a symbol "+" (assumed by default), "-", or "="— respectively meaning that the triples with this object should exist, should not exist, should be the only that exists for that subject and predicate. For syntactic convenience, we also allow a pattern to consist solely of the source, meaning no further constraint on its triples is imposed. Additionally, we sometimes use as filter for a subject `?LSA` an expression of the kind "`?LSA clones ?LSA2`", meaning that `?LSA` should have the same content of `?LSA2` plus additional constraints specified by any following triples. The above definition of a pattern naturally induces the concept of an LSA matching a pattern (modulo a substitution of variable to terms).

[1]A bnode, or blank node, is an locally scoped identifier. Bnodes are used to represent structured property values within LSAs, although we do not elaborate this concept within this paper.

The semantics of an eco-law is then as follows. It consumes a set of *reactant LSAs* based on left-hand side patterns and produces a set of *product LSAs* based on the right-hand side patterns. In particular, right-hand side patterns are to be seen as post-conditions applied to the selected reactant LSAs. Eco-laws also obey a numeric transformation rate `r` representing a Markovian rate in a continuous-time Markov chain (CTMC) system. Such a rate can be omitted, in which case it is assumed to be infinite, that is, the eco-law is executed with "as soon as possible" semantics.

An eco-law can apply in many different locations of the ecosystem, and to different sets of (co-located) LSAs. We call *reaction* the pair consisting of a set of reactant LSAs and their corresponding product LSAs that an eco-law can trigger. Execution of a reaction amounts to atomically removing reactant LSAs from the LSA-space and inserting product LSAs back.

As an example eco-law, consider the following, which aggregates two LSAs produced by a crowd sensor, so as to keep the most recent:

```
?LSA  eco:type msm:crowd; msm:time =?T; +
?LSA2 eco:type msm:crowd; msm:time =?T2(?T2<?T);
--r-->
?LSA
```

Note that in the right-hand side we do not specify triples for `?LSA2`, which means that the LSA with id `?LSA2` will be removed, and that for `?LSA` we simply state it will be left unchanged. Another example of an eco-law, used to make a display activate and show an advertisement as soon as the presence of a person with a matching profile is sensed (e.g., the LSA of an user present in the same space), is the following:

```
?DIS eco:type msm:display; msm:status ="ready";  +
?ADV eco:type msm:ad; msm:content ?C;            +
?USR eco:type msm:usr; msm:prof ?P(?P matches ?C);
--r-->
?DIS msm:status ="showing"; msm:service ?ADV; +
?ADV +
?USR
```

Note that object `="ready"` in the left-hand side means that `"ready"` is the only object for subject `?DIS` and predicate `msm:status`, while object `="showing"` in the right-hand side means that `"showing"` should replace any previous value, while using object `"showing"` (or `+"showing"`) would mean adding value `"showing"`. We have defined a formal mapping between eco-laws and SPARQL/SPARUL, which is not reported here for brevity. As an example, the latter eco-law is written as:

```
SELECT DISTINCT * WHERE{
  ?DIS eco:type msm:display .
  ?DIS msm:status "ready" .
  FILTER NOT EXISTS { ?DIS msm:status ?o .
                      FILTER (?o!= "ready") }
  ?ADV eco:type msm:ad .
  ?ADV msm:content ?C .
  ?USR eco:type msm:user .
  ?USR msm:prof ?P; FILTER(?P rdf:type ?C) .
}
REMOVE DATA {!DIS msm:status ?o}
INSERT DATA {!DIS msm:status "showing"}
INSERT DATA {!DIS msm:service !ASV"}
```

Put in more general terms, each eco-law is mapped to one single SPARQL `SELECT` query, and a sequence of SPARUL `REMOVE` or `INSERT` statements. The first query checks whether and how the eco-law applies, yielding a set of bindings for all the variables involved—one binding of variables per each solution found, namely, per each set of reactant LSAs. Given one binding, the SPARUL statements are used to apply the eco-law. To this end, we let a placeholder `!VAR` stand for the value to which variable `?VAR` is linked to by the binding produced by SPARQL query.

C. Architectural Components

From an implementation viewpoint, the framework can be realised as a lightweight and minimal middleware that reifies LSAs in the form of semantic tuples, to be dynamically stored and updated in a system of spatially-situated tuple spaces spread over the devices of the network. The eco-laws governing the ecosystem are deployed in all network nodes, and apply locally[2]. Each node comprises a set of modules managing LSAs and Eco-laws, described in turn, all based on functionality provided by the ARQ query engine [1] and Pellet reasoner [17].

External Interface: The interface by which agents, devices, services and other nodes – namely, the *external environment* – interacts with the node providing operations to locally `inject` a new LSA, `observe` an LSA with a known ID, and `modify` and `remove` the LSAs the agent owns (injected).

Space: A space represents a passive component, similar to a tuple space, storing LSAs that are local to the node. It is responsible for the identification of LSAs within the node, hence, it manages LSA naming through unique identifiers. It is also the module in charge of implementing any possibly sophisticated indexing algorithm and data structure with the goal of quickly retrieving and filtering candidate LSAs matching an eco-law to be executed. Considering the above-mentioned technologies, the space is easily realised as an RDF-store.

Matcher: During the processing of eco-laws, the reaction manager checks whether an eco-law can apply to a candidate set of LSAs—extracted from the space based on pattern matching. More specifically, it computes all the bindings for a given

[2]At the time of writing, one such prototype is under construction in the context of SAPERE project [2].

eco-law by executing the SPARQL query. Additionally, this component also computes all filter expressions, the evaluation of which is deferred to Pellet, which – other than standard mathematical functions – allows one to code external functions computing, e.g., any ad-hoc matching between the arguments. The semantics provided by RDF Schema (RFDS) [8] and the Web Ontology Language (OWL) [9] support vocabularies that define classes of resources, semantically-rich relations, and sets of restrictions on how both may legally be combined. Application of these vocabularies to an RDF model may be verified for correctness, and inferences – such as the classification of resources – may be drawn. Indeed, standard OWL classification provides one approach to realising ontology-based semantic matching in our eco-law language. A filter `?A matches ?C` can be interpreted as `?A rdf:type ?C`. In standard OWL semantics, this considers an individual a valid substitution for `?A` if its description satisfies the set of restrictions that describe `?C`, an OWL Class description. Hence, whenever one or more ontologies are used in the pervasive ecosystem, they must be accessible to the Matcher component.

Reaction manager: The reaction manager handles events occurring within the node, based on the set of eco-laws it holds. An event describes either an external operation upon an LSA (injection, observation, removal or modification), or the activation of a specific eco-law to be executed. Events of the first kind are considered with highest priority, and are simply processed by interaction with the space. In the second case, the event is characterised by a reaction, indicating the eco-law it refers to, its binding, and the time at which it should be executed. The reaction manager exploits a simple *scheduling engine*, maintaining a list of scheduled events sorted by their occurrence time. It takes the next event, and executes the corresponding SPARUL statements through ARQ.

IV. Adding spatial computing features

The framework described so far contains only interactions of agents working in the same node, mediated by some form of local knowledge, similarly to other coordination frameworks like those in [14], [7]. In this section we develop an arguably minimal extension, enabling the possibility to enact spatial computing, by which distributed behaviour useful to pervasive computing purposes can be supported. This is based on three ingredients we describe in turn.

LSAs reifying location: We shall assume that each LSA carries one property named `eco:#loc`, holding one URI value representing the ID of the location (i.e., the node) in which the LSA currently resides. We refer to this property as a "synthetic" one, for it is not specified by the agent that injects the LSA, but it is rather created and maintained by the infrastructure. Additionally, this is a property that cannot be changed by agents, but only by eco-laws as we will detail in the following.

Space-time LSAs: A core idea of pervasive service ecosystems is that, all information deemed important for the overall system coordination should be reified within LSAs. This applies, in fact, to all agent activities, and the data, knowledge, and events they produce. Additionally, contextual information bridging ecosystem evolution with the physical word has to be properly reified into what we call synthetic LSAs—again, "synthetic" here refers to the fact that these LSAs are maintained by the infrastructure (e.g., by some agents in charge of injecting them and keeping them updated by some timing policy). Most importantly here, this concerns the space-time situation of the computation, that is, at which time we are currently executing, and how the local space is shaped.

Accordingly, we shall first assume that in each node we have the so-called *time LSA*, an LSA carrying information about the current time in the node, which is of the kind:

```
lsa:timelsa321
   eco:type eco:#timeLSA;
   eco:#time "2011-05-30T11:00:00";
   eco:#loc sid:node34164@room132;
```

Concerning space, we reify the shape of space as can be perceived from a single node, namely, what are the neighbours in the node's proximity (possibly including their IDs, their estimated distance, the kind of connectivity, the maximum communication bandwidth, the relative orientation in space, and any other information the infrastructure can discern). In particular, in any node, we assume that for each neighbour there is a synthetic *neighbour LSA* of the kind

```
lsa:neighlsa456
   eco:type eco:#neighbourLSA;
   eco:#loc sid:node34164@room132;
   eco:remotelocation sid:node34163@room132;
   eco:distance "51.3";
   eco:orientation "north-east";
```

stating that the neighbouring node is at location `node34163@room132`, which is at expected distance 51.3 (meters) in a north-easterly direction.

We reiterate that the connection between such LSAs and the neighbours they represent is entirely implicit; nodes do not directly manage their remote representations, and these synthetic LSAs are not proxies through which information from remote spaces may be obtained.

Relocation service: One of the motivations for reifying these synthetic LSAs is the ability, by means of proper eco-laws, to make their actual content impact ecosystem dynamics, by which we can fully achieve context-dependent behaviour. Concerning space, for instance, one can develop an eco-law that changes the value of the `eco:#loc` property of an LSA, replacing it with that of a neighbouring node. One such eco-law is:

```
?LSA eco:type msm:crowd; eco:status ="tomove"; +
?NEI eco:type eco:#neighbourLSA; eco:remote ?L;
--r-->
?NEI + ?LSA eco:#loc =?L; eco:status -"tomove";
```

This causes an LSA to have a location that no longer fits the current node in which it resides. A lower-level middleware service then, can be in charge of intercepting such LSAs before they are injected in the space, and relocating them to the proper neighbour, by an asynchronous request.

Note that our management of space-time aspects has the advantage of being fully declarative—e.g., an LSA with a new location can just be perceived as the LSA having been relocated. By this approach we retain the spatial locality property of eco-laws, mitigating the need to perform synchronisation across spaces during their application. This approach also orthogonally supports more advanced concepts of "topology", such as notions of social neighbouring (connecting smartphones of people who are friends on a social network as envisioned in [16]).

V. Examples

While a basic use of the above ingredients supports relocating information to mirror its physical counterpart's movements, they also permit the enactment of some patterns of spatial computing that enable useful interactions within pervasive service ecosystems; here we describe 4 such patterns, accompanied by illustrative examples.

A. Gossiping

We first show how we can make an LSA spread from a given location to all the nodes of the network, with the further ability of expiring everywhere at a given time. This is the set of eco-laws realising this behaviour:

```
[GOS] % ?GOS gets spread in any neighbour
?GOS spc:type spc:goss; +
?NEI eco:type eco:#neighbourLSA; eco:remote ?L;
--r-->
?NEI + ?GOS + ?CLO(?CLO clones ?GOS) eco:#loc =?L;

[AGG] % Of two similar LSAs, it removes one
?AGG spc:type spc:aggr; spc:content ?C; +
?AG2 spc:type spc:aggr; spc:content ?C;
--> ?AGG

[DEL] % Disposes an LSA if its deadline expired
?DEL spc:type spc:del; spc:deadline = ?T;    +
?TIM eco:type eco:#timeLSA; eco:time ?T2(?T<?T2);
--> ?TIM
```

Initially, a gossip LSA with `spc:type` set to values `spc:goss`, `spc:aggr`, and `spc:del` is injected in a node—so that all three eco-laws apply. The former eco-law makes any gossip LSA `?GOS` (having property `spc:type` set to `spc:goss`) create a cloned version `?CLO` relocated in a neighbour node `?L`. Iterative application of this eco-law over time (at rate r) and over all nodes makes copies of `?GOS` flood the network. The second eco-law takes two gossip LSAs (of kind `spc:aggr`) with same content (namely, relative to the same source) and drops one: this is used to avoid multiple versions residing in the same node. Finally, the latter eco-law fires when the current time in a node is greater than the deadline time `spc:deadline` defined in the gossip LSA, causing removal of that LSA. Of course, rate r is to be properly designed to tune the network load, using considerations similar to those discussed in [18]. Note that the three eco-laws orthogonally apply—one could leverage spreading without time-disposal, or time-disposal alone, and so on.

In the context of pervasive computing applications, this pattern can be useful to advertise an event happening in a given node to the whole network. Considering e.g., the application scenario in [12], it could be used to advertise a fire alarm in an exhibition centre, so as to immediately trigger all the activities necessary to safely steer people towards exits.

B. Gradient

A variation of the above diffusion mechanism can be used to create a gradient data structure—a key brick of several spatial computing patterns [18], [4], [19]. This is based on the idea of spreading copies of an LSA such that each of them holds the estimated distance from the source according to the shortest path. We shall also equip each LSA with a reference to the next node to traverse in order to follow the gradient towards the source, and provide a mechanism to limit the gradient to a given spatial extent (gradient horizon). This is the set of eco-laws realising this behaviour:

```
[GRA] % ?GRA gets spread with increasing spc:dist
?GRA spc:type spc:gra; spc:dist ?D;
     spc:rng ?R; eco:#loc ?LG; spc:source ?S +
?NEI eco:type eco:#neighbourLSA; eco:remote ?L;
     eco:distance ?D2(?D+?D2<=?R);
--r-->
?NEI + ?GRA +
?CLO(?CLO clones ?GRA) eco:#loc =?L; eco:prev =?LG;
     spc:dist =?D3(?D3 = ?D+?D2);

[SHR] % Of two paths, the shortest one is kept
?GRA spc:type spc:short; spc:content ?C;
     spc:dist ?D; +
?GR2 spc:type spc:short; spc:content ?C;
     spc:dist ?D2(?D2>?D);
--> ?GRA
```

Initially, a gradient LSA with `spc:type` set to `spc:gra` and `spc:short`, and `spc:dist` set to 0 is to be injected in a node; we also assume (for the sake of the following examples) that property `spc:source` is initially set to the LSA-id itself—by spreading, this property will hold everywhere the ID of the LSA from which the gradient originates. The former eco-law creates a cloned version `?CLO` relocated in a neighbour node `?L`, with an increased value of distance depending on the estimated distance of that neighbour—note this does not happen if horizon `?R` is escaped. The second eco-law takes two gradient LSAs with same content and keeps the one with smaller distance.

In the context of pervasive computing applications, this pattern can be useful to advertise an event in a given node with additional information on how its source can be reached. E.g., as a fire alarm has been spread, people can be directed

to any exit (acting as gradient source) by signs appearing in their smartphone or on public displays, properly reflecting the direction to take as can be inferred from property `eco:prev` of gradient LSAs.

C. Partitioning

This pattern is used to partition a network in n areas once n nodes (sources) have been selected as candidate centres of these areas, ensuring that these areas have also similar size whenever possible. This is achieved by making each source spread a different gradient, such that propagation does not overlap in nodes in which other gradients are already established with smaller distance to another source. In this way, each node will belong to the area of the nearest source node. We observe that the above eco-laws for gradients already support this behaviour, provided that the n source LSAs share the same `spc:content`, but have e.g., different values of a property `spc:area`.

An example application can be envisioned in the context of adaptive pervasive displays [18]. Assuming we have n different advertisements to show in an airport, and we want the set of displays showing one of them to form a contiguous area (to avoid people perceiving different advertisements as the pass by neighbouring displays), we can use the partition pattern and let displays choose what to visualise depending on the area they belong to. Note this pattern automatically accommodates the injection and removal of sources.

D. Path

Another pattern proposed in [11] is the path connecting two distinct nodes, possibly enlarged to the set of devices whose distance from that path is smaller than a given horizon h. This is achieved by first making the two nodes (called source and target) create their own gradient (with sufficient horizon to reach each other). As soon as the source perceives the target, it should gossip a new LSA indicating their relative distance d. Then, each node should compare d with the sum of its distance to source and target: if the difference is smaller than h, then a new LSA is created to tag this node as being part of the path. The eco-laws realising this behaviour are as follows:

```
[PATH] % SRC senses TRG gradient, gossiping distance
?GRA spc:type spc:gra spc:pathtrg;
     spc:dist ?D; spc:source ?TRG; +
?SRC eco:type spc:gra spc:pathsrc; spc:dist 0;
--r-->
?GRA + ?SRC +
?GSP(?GSP clones ?GRA)
     eco:type =spc:goss =spc:aggr;
     spc:pathtrg =?TRG; spc:pathsrc =?SRC:

[SUM] % Reifying ?PTH if the node is inside the path
?TRG spc:type spc:gra; spc:dist ?DT; +
?SRC spc:type spc:gra; spc:dist ?DS; spc:rng = ?R +
?GSP spc:pathtarget =?TRG; spc:pathsrc =?SRC;
     spc:dist ?DP(?DP>?DT+?DS-?R);
--->
?TRG + ?SRC + ?GSP +
?PTH(?PTH clones ?GSP) spc:type =spc:path =spc:shr;
```

The former eco-law makes a source detect the target at distance `?D` and correspondingly gossip (without timeout) that value of distance. The second eco-law makes any node inside the path creating an LSA of kind `spc:path`. Note the latter LSA is also of type `spc:shr`, so that only the copy with shortest distance is kept.

This pattern can be useful to mark the transiting area of people steered from a place to another in an articulated environment, as in the case of people moving from one gate to another in an airport. Public displays can be programmed to show signs towards the target gate only if they stay inside the path dynamically computed as described above. In this way, we avoid affecting all the displays of the airport, but may still handle the case of people departing slightly from the optimal path.

VI. Conclusions and Future Work

In this paper we described how spatial computing concepts can be injected into a framework of pervasive service ecosystems, in terms of a minimal set of features concerning reification of space-time information (namely, bridging the gap between the computational world and the physical world), and a relocation service designed to remedy mismatches between the locality declared by some information and its actual position. Several examples of applications to spatial computing structures are provided to explain the details of these mechanisms and show the usefulness in pervasive computing. We believe that the work presented in this paper has a validity beyond the pervasive ecosystem framework, for it can be smoothly applied to any distributed system based on a notion of shared data-space (contrasting approaches based on message-passing such as e.g. [11]).

The proposed chemical model can be extended in several ways, all of which will be subject of our future investigations. First of all, we currently retain a quite rigid structure in which the number of reactants and products is statically defined: further studies are needed to understand to which extent the corresponding language can mimic transformations working over sets of LSAs whose size is not known a priori. Then, we currently rely on CTMC semantics to trigger eco-laws, without considering further priority constructs which could be interesting. Also, we assume a flat set of LSAs without any hierarchical structuring of the topological space, which would be of some interest for pervasive computing applications. Finally, we note that a distributed setting requires security and privacy mechanisms that complement the openness of the system. We intend to develop such mechanisms as part of the ecosystem fabric: for example, supporting spatially restricted information flow and using inference based on property and concept hierarchies to appropriately abstract information viewable by agents. Alternatively, general frameworks tackling security in coordination models like [13] can be evaluated.

A further roadmap for future works aimed at strengthening the relationships between spatial computing, space-based coordination models, and their applicability to pervasive computing applications, includes the following activities: *(i)* identifying a

concept of expressiveness of spatial computations and a minimal set of mechanisms to achieve it, along the lines of [5]; *(ii)* studying techniques for predicting and controlling the global behaviour that emerges out of the local coordination rules; *(iii)* deepening the advantages of using semantic matching in the context of spatial computing patterns, as e.g., exploited in [15]; *(iv)* thoroughly analysing the applicability of spatial computing to emerging ICT scenarios like smart cities, intelligent traffic control, and augmented social reality.

ACKNOWLEDGMENTS

This work has been supported by the EU FP7 project "SAPERE - Self-aware Pervasive Service Ecosystems" under contract No. 256873.

REFERENCES

[1] ARQ - a SPARQL processor for Jena. http://jena.sourceforge.net/ARQ/, 2011.

[2] Self-aware pervasive service ecosystems. http://www.sapere-project.eu, 2012.

[3] J.-P. Banâtre and T. Priol. Chemical programming of future service-oriented architectures. *JSW*, 4(7):738–746, 2009.

[4] J. Beal. Flexible self-healing gradients. In *Proceedings of the 2009 ACM Symposium on Applied Computing (SAC)*, pages 1197–1201. ACM, 2009.

[5] J. Beal. A basis set of operators for space-time computations. In *Self-Adaptive and Self-Organizing Systems Workshop (SASOW 2010)*, pages 91 –97, sept. 2010.

[6] T. Berners-Lee and D. Connolly. Notation3 (N3): A readable rdf syntax. W3C team submission, W3C, 2011. http://www.w3.org/TeamSubmission/n3/.

[7] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.

[8] R. V. Guha and D. Brickley. RDF vocabulary description language 1.0: RDF schema. W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-schema-20040210/.

[9] M. Krötzsch, P. F. Patel-Schneider, S. Rudolph, P. Hitzler, and B. Parsia. OWL 2 web ontology language primer. Technical report, W3C, Oct. 2009. http://www.w3.org/TR/2009/REC-owl2-primer-20091027/.

[10] E. Miller and F. Manola. RDF primer. W3C recommendation, W3C, Feb. 2004. http://www.w3.org/TR/2004/REC-rdf-primer-20040210/.

[11] MIT Proto. software available at `http://proto.bbn.com/`, Retrieved Nov. 1st, 2010.

[12] S. Montagna, M. Viroli, M. Risoldi, D. Pianini, and G. Di Marzo Serugendo. Self-organising pervasive ecosystems: A crowd evacuation example. In *Workshop on Software Engineering for Resilient Systems*, volume 6968 of *LNCS*, pages 115–129. Springer, 2011.

[13] A. Omicini, A. Ricci, and M. Viroli. An algebraic approach for modelling organisation, roles and contexts in MAS. *Applicable Algebra in Engineering, Communication and Computing*, 16(2-3):151–178, Aug. 2005.

[14] A. Omicini and F. Zambonelli. Coordination for Internet application development. *Autonomous Agents and Multi-Agent Systems*, 2(3):251–269, Sept. 1999.

[15] D. Pianini, S. Virruso, R. Menezes, A. Omicini, and M. Viroli. Self organization in coordination systems using a WordNet-based ontology. In *4th IEEE International Conference on Self-Adaptive and Self-Organizing Systems (SASO 2010)*, pages 114–123. IEEE CS, 27 Sept.–1 Oct. 2010.

[16] A. Rosi, M. Mamei, F. Zambonelli, S. Dobson, G. Stevenson, and J. Ye. Social sensors and pervasive services: Approaches and perspectives. In *PerCom Workshops*, pages 525–530. IEEE, 2011.

[17] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5:51–53, June 2007.

[18] M. Viroli, M. Casadei, S. Montagna, and F. Zambonelli. Spatial coordination of pervasive services through chemical-inspired tuple spaces. *ACM Transactions on Autonomous and Adaptive Systems*, 6(2):14:1 – 14:24, June 2011.

[19] M. Viroli, D. Pianini, and J. Beal. Linda in space-time: an adaptive coordination model for mobile ad-hoc environments. In M. Sirjani, editor, *Proceedings of the 14th Conference of Coordination Models and Languages (Coordination 2012),Stockholm (Sweden), 14-15 June*, Lecture Notes in Computer Science. Springer, 2012.

[20] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson. Pervasive ecosystems: a coordination model based on semantic chemistry. In S. Ossowski, P. Lecca, C.-C. Hung, and J. Hong, editors, *27th Annual ACM Symposium on Applied Computing (SAC 2012)*, Riva del Garda, TN, Italy, 26-30 March 2012. ACM.

[21] M. Viroli, F. Zambonelli, G. Stevenson, and S. Dobson. *From SOA to Pervasive Service Ecosystems: an approach based on Semantic Web technologies*. IGI Global, 2012. Available for reviewers to download at: http://apice.unibo.it/xwiki/bin/download/Publications/SemanticSapereIGI2012/chapter.pdf.

[22] F. Zambonelli and M. Viroli. A survey on nature-inspired metaphors for pervasive service ecosystems. *International Journal of Pervasive Computing and Communications*, 7(3):186–204, 2011.