Presented in special session at Intelligent Environments 2005 (IE'05), Colchester, UK, 28-29 June 2005

Deus Ex Machina: Engineering Emergence in an Amorphous Computer

Adam King, Graham Clarke, Victor Callaghan

Department of Computer Science, University of Essex, United Kingdom
vic@essex.ac.uk

Abstract. An amorphous computer is a multitude of tiny computers each with a CPU, memory, and local communication capability. An iSurface is a particular instance of an amorphous computer, an "intelligent" coasting capable of computing, sensing and some limited output. This can be regarded as a massively parallel computing system, albeit one with random structure and unreliable nodes, and, as such, is likely to be extremely difficult to program conventionally. Prior research into amorphous computing suggests that utilising principles of self-organisation is successful in providing some functionality to an amorphous computer. Here we argue there is a definite need for techniques in engineering emergent functionality to match user requirements; the development of systems that utilise self-organisation in a user-directed manner. We present an early version of a methodology to do this and an example of its successful usage.

1 Introduction

The vision of ubiquitous, ambient or pervasive computing conjures up a world in which the whole environment is filled with networked artefacts with computational abilities. Some artefacts may be visible but many would be invisible to the casual observer. In order to be able to fully exploit this ambient computing power potential we require some manner of making these devices work together without explicit external intervention being a regular occurrence. This means that we need to explore the possibility of self-organisation among large populations of communicating devices. This form of self-organisation will have to be provided with a goal or set of goals explicitly or implicitly in order for it to be able to orient itself appropriately. One way in which this might work is if there was available to the devices a representation of the sorts of collective activities that they might perform and a description of the sorts of organisation they would have to achieve to be able to do this adequately. Such representations might be called ontologies specifying the varieties of combinations and their function that any particular set of devices might achieve collectively. An alternative possibility is that the user through some relatively simple interface provides the intention or goal and then the devices achieve some degree of self-organisation based upon this. The drawback with the former approach is that the ontologies will never be exhaustive and with the latter approach that a degree of sophisticated interaction is required from the user. There will undoubtedly

be other approaches that may take their inspiration from biology and involve something like an artificial life approach to this sort of self-organisation, the biological world being by far the most successful example of this sort of process we know of.

With the advent of nano-technology the possibility of producing a sort of parabiological world presents itself and, short of finding ways to actually grow computing systems using biological material, this is probably the lowest level at which we will be able to explore this possibility of large populations of agents co-operating to achieve a task. This apparently different problem is connected with the first in that both involve a large number of communicating computers capable of carrying out some sophisticated operations locally but potentially capable of achieving more by co-operation with a network of others.

We believe that the study of co-operating devices at the nano level is synergetic with the problem of communication and co-operating between devices at the micro and macro scale and it is with this in mind that we are looking to investigate an approach to the problem using intelligent agents and limited capacity nano-scale devices. At the current stage these are simulated but there is no reason why they might not operate in the real world in the near future.

"Today, people's domestic spaces are becoming increasingly 'decorated' by electronic or computer based artefacts (gadgets) varying from mobile phones, through CD players, to transport systems and beyond." [1].

This quote by Callaghan et al conjures the vision of an environment utterly saturated with these gadgets. This environment is not static however; it is hugely dynamic with gadgets appearing and disappearing, and moving through this "cloud of intelligent artefacts" (a concept explored in [2]).

The idea of an intelligent cloud is an ideal starting point for discussing this research. The reader is invited to consider a cloud; a large amorphous collection of water molecules naturally following self-organising principles. Now consider a massive collection of devices arranged in an ad hoc network that may vary with time. Trying to associate these devices by hand would be ridiculous, the combinatorics involved renders the task all but impossible. There is a serious problem of overload.

Now, clouds have no external force associating each molecule to its neighbours. Water molecules are attracted to fellows and link in a demonstration of self-organisation.

These same principles could be applied to these device-saturated environments, treating them as massive self-organising amorphous systems, removing the need for human intervention.

1.1 Amorphous Computing

An Amorphous Computer, as defined by [3], is a multitude of identical tiny computers (known as particles) each with a CPU, memory and communication capability. These particles are able to sense (undefined) aspects of the environment and may perform (again undefined) actions upon the environment. It is a personal

opinion that these "particles" may be any device of any size with any capabilities, but Katznelson's definition is more relevant for the purpose of this research.

The field of Amorphous Computing is the development of organisational principles and programming languages for obtaining coherent behaviour from the interaction between large numbers of unreliable particles that are interconnected in unknown, random and time-varying ways.

An amorphous computer can be regarded as a massively parallel computing system [4]. Previous investigations into massively parallel systems, such as cellular automata, are one source of ideas for dealing with amorphous systems but another source of ideas is research into self-organising systems. Abelson et al, following the self-organising route, suggest that a possible solution to the challenge is to embed all the required code into the particles at the time of manufacture. With an amorphous computing system where each particle has this code preloaded, the particular functions that are activated within any particle depends on the messages it receives from the local environment. The principle means of interpreting messages propagated across the particles are based upon biological mechanisms. In the case of fixed particles in an amorphous computer a gradient field (like a diffusion of chemicals) is established based upon the messages being received locally. In general a gradient field spreads out from a specific source and falls off in concentration with distance from that source. So for a specific particle the "concentration" (the number of hops the message has made) and "type" of messages impinging upon the particle will trigger specific processes. This biological metaphor is indicative of the approach taken by Abelson et al, programming an amorphous computer via wave propagation; an "anchor" particle broadcasts a message to its neighbours who, in turn, broadcast to their neighbours and thus the message diffuses throughout the system (cf. pheromone or tropism). These waves can have a controlled size and can thus define a region that can be used to inhibit the growth of another region. It is these message waves that select the operation of a particle.

Butera [5] expands upon this by arguing that attempts to produce robust general methods for programming massively parallel systems have been unsuccessful. It is therefore likely that an amorphous computer, which, as we have already observed, can be seen as a massively parallel system, is likely to be extremely difficult to program conventionally. The specific issues that make this a hard problem are asynchronous operations and communication, managing faults in imperfect elements, establishing network position and structure, being limited to communications with local elements, predicting the range of functions that must be pre-coded and the vast number of possible states the system can be in.

Butera claims that it would never be practical to expect a human to be able to structure a procedure for unrestricted, flexible use on such a system. Assuming this is true, he argues that, instead of getting a human to structure these procedures, it is necessary to get the procedures to structure themselves. Butera proposes the theory that: "A programming model employing a self-organising ecology of mobile process fragments supports a variety of useful applications on a paintable computer." [5]. In support of this statement, Butera offers a distributed programming methodology, known as 'process self-assembly', which maps existing techniques in material and virtual self-assembly to an amorphous computer. In addition, a programming model

is introduced, as is the "process fragment" - the atomic element of process self-assembly. The whole project demonstrates the feasibility of a mobile agent paradigm.

1.2 iSurfaces

An iSurface (intelligent surface) is a specialisation of the amorphous computer concept, a tow-dimensional layer of particles painted over the surface of an object; i.e. a wall, desk, or a face of a mobile phone or laptop computer. Turn a wall into an iSurface and you can use it as a television; change your "wallpaper" design on a whim; use the wall to determine concentrations of light or temperature within a room and turn on the necessary lights or heating elements; treat parts of the wall as keyboards or touch-screen; possibly even to record or produce sound. If you wanted to transfer data you could have a fingertip iSurface. Place this against the wall and download the data. Head to another surface and upload with a touch. This is a vision of near complete integration of environment and "intelligent" device, nearly anything can be given "intelligence" with just a coat of paint.

1.3 iCells

iCells are the particles of an iSurface; these are theoretical nano-scale self-contained computing units with attached sensors (touch, light, sound, temperature, humidity, gas, and a camera) and actuators (a light emitting device and a light manipulating antenna). The iCell has enough memory and processing capabilities to handle anything (that is, we don't need to worry about program, message, and data size inside an iCell; the assumption is that anything can be stored or executed) and it runs Java. It also has a short-range communication ability that operates on an undirected broadcast basis. All surrounding iCells within range can receive this signal without chance of error. Power is not an issue in this case.

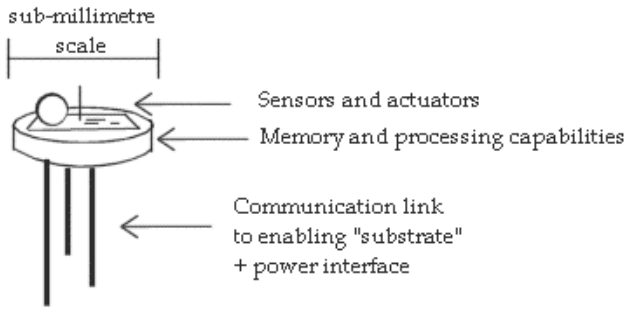

Fig. 1. Conceptual diagram of an iCell

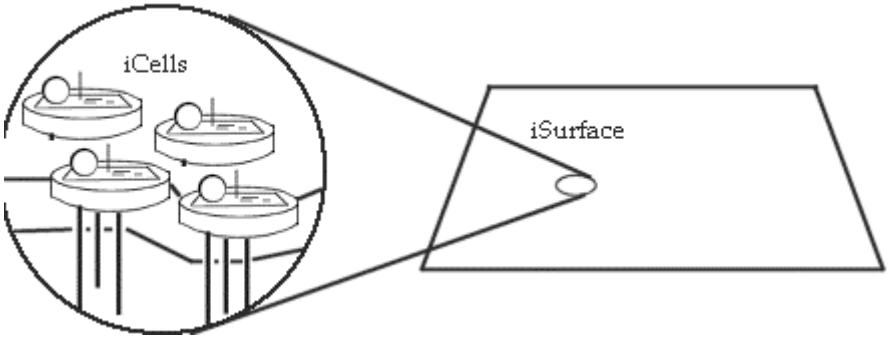

Fig. 2. Cutaway conceptual diagram of an iSurface. The iCells are all embedded into a communications enabling substrate

1.4 Programming iSurfaces

It is known that the "conventional" amorphous computing approach relies on pre-determined functionality and this is "hard-coded" (via the self-organising growing point language) into the elements. Only sections of this code are active at a given time however; these are defined by the messages received by the particle. Paintable computing provides an alternative, programming an amorphous computer via use of self-organising mobile code. However, the description of this approach simply has code entering the system from an unexplained source leading one to conclude that either a user is entering appropriate code but isn't mentioned, or that no user-intervention takes place.

It is felt that, due to its limited predetermined functionality, the "conventional" amorphous computing approach fails to exhibit the flexibility one might expect from an intelligent surface; the ability to determine / alter functionality of an iSurface at runtime by a user. The paintable computer approach does go some way towards rectifying this, providing a mechanism for an amorphous computer that is more "dynamic" in terms of the range of functions it can perform. Unfortunately, the paintable computer approach fails to answer several vital questions.

Firstly, how do we decide on the agents necessary for a specified functionality?

Secondly, how do we decide how these agents interact?

Finally, how do we ensure the system is structured as we desire it to be?

1.5 Deus Ex Machina Approach

These unanswered questions show that there is a definite need for techniques in engineering emergent functionality to match user requirements; the design / selection / insertion of agents that utilise the self-organising principles of amorphous computing in a user-directed manner. Essentially this would revolve around a methodology for decomposing system design into component mobile agents; the two major challenges of which would be the determination of minimal agents needed (potentially even

identifying agents common across several applications) and the determination of how these agents must interact locally in order for global functionality to emerge. It would be possible to couple such a methodology with the ability for the user to "dictate" agent structure; that is, agents are explicitly placed in the iSurface by a user either to target specific particles or to act as a "seed" in order to propagate like a virus across the surface. This research investigates the hypothesis that such a "Deus Ex Machina" approach of adding user-selected & designed combinations of agents, but still employing principles of self-organisation, will allow a far quicker and more precise method of engineering functionality in an amorphous computer.

1.6 Structure of this Paper

This paper introduces the first version of the Deus Ex Machina methodology for engineering emergent behaviour in an amorphous system such as the iSurface.
Firstly we shall discuss what we mean by emergence and previous attempts to define, measure and engineer it. Following on we present the initial version of our methodology describing the simulator system used to develop an iSurface and perform experiments. We will also discuss how to evaluate an iSurface's functionality using this system. Finally we will examine an initial experiment involving a very simple iSurface application and how the methodology aids in its construction.

2 Emergent Phenomena

Before going further with this paper we need to tackle some issues that often cause much disagreement; namely definitions of what we mean by emergence, complex systems and self-organisation. Everyone has their own ideas about these concepts and it is recognised that people are unlikely to change their opinions. Therefore, the definitions provided here are just for consideration in the context of this paper and research.

Darley [6] suggests that the defining characteristic of complex system is that some of its global or macro behaviours, which are the result of interactions between a large number of relatively simple parts, cannot be predicted simply from the rules governing these interactions. He introduces the concept of an "emergent phenomenon" as large-scale group behaviour of a system that doesn't seem to have any clear explanation in terms of the system's constituent parts. There doesn't appear to be any discernible difference between Darley's usage of the terms "complex system" and "emergent phenomenon".

In a similar vein, Bishop & Potgieter [7] make the assertion that a multi-agent system in which the agents communicate with each other using predefined protocols is a complex system. An emergent system, referred to as a complex adaptive system, is characterised by complex behaviour as the result of interactions between system components and the environment.

These two attempts to define, and differentiate, complex systems and emergent systems end up producing near identical definitions for both concepts, thus hinting at the widespread disagreements about these concepts.

Johnson [8] proposes a definition that is in line with the concepts used in this research. He suggests that a complex system is any system composed of multiple agents interacting in multiple ways, following local rules and oblivious to any higher level instructions. An emergent system is a complex system but with a discernible "global" behaviour, only truly observable when the system is considered as a whole.

In addition to this definition of complex & emergent systems we briefly define another couple of subsets of complex systems.

A self-organising system is a complex system where the interactions between agents cause the system to tend towards a stable global / macro state.

A chaotic, or non-deterministic, system is a complex system where uncertainty in state increases with time, making long-term prediction of behaviour impossible.

A Venn diagram (fig 3) may be used to present the set of complex systems.

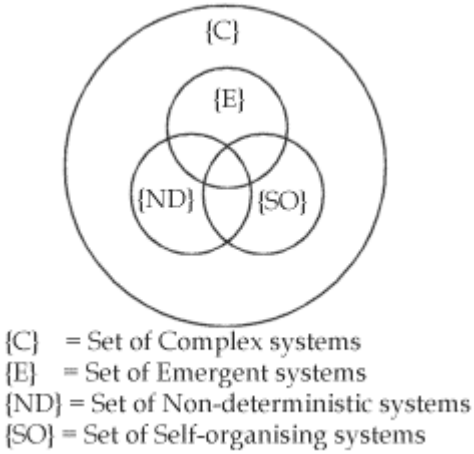

Fig. 3. Venn diagram showing relations between classifications of complex systems

2.1 Determination of Emergence

Still the concept of emergence is a little vague. How do we decide when a macro behaviour is emergent rather than explicitly programmed into the elements of the system?

Ronald et al [9], taking the Turing Test as inspiration, devised an emergence test in terms of an observer's claimed inability to drive observed global behaviour from their knowledge of the local interactions. In short, if the observer is surprised by the results

then it is emergent behaviour. This test is described in terms of the following three conditions.

Design
- The designer has constructed the system by describing local interactions between components, in a language L1.

Observation
- The observer is fully aware of the design, but describes global behaviours and properties of the running system, over a period of time, using a language L2.

Surprise
- The language of L1 & L2 are distinct and the causal link between the elementary interactions programmed in L1, and the behaviours observed in L2 is non-obvious to the observer, who therefore experiences "surprise".

Note that the label "Emergent" relies on an observer's understanding of the system, or rather the lack of it. Despite the attempt at formalisation, the emergent test is still utterly subjective; one observer may claim emergence, another may claim to understand perfectly why something is happening.

Maybe a different approach is needed. Every time we go up a level of abstraction we observe new behaviours in the level below us. Therefore, instead of trying to prove emergence, attempt to disprove it.

Assume that a behaviour is emergent. Is the macro behaviour simply a scaled up version of a micro behaviour? If so, then the "emergent" label is lost. Are the agents following local rules to interact with others, or are their connection specified by an external agency on the same level of abstraction as us rather than a result of situatedness in the environment? If the latter is true then the "emergent" label is lost. This approach will be explored further later on in this paper.

3 Constructing Emergent Systems

Having covered what emergence is (along with other forms of complex systems) and briefly discussed how to determine if a system has emergent properties, we now examine existing methods of deliberately creating a system that exhibits some desired emergent behaviour.

We'll begin with looking at designing behaviour-based systems, traditionally a "showcase" for emergent phenomena, and a well-established field.

Brooks, a pioneer of these systems, states [10] that in all engineering endeavours, it is necessary to decompose a complex system into parts, build the parts, the interface them into a complete system.

He suggests that this could be accomplished in two ways. Firstly, decomposition by function. Prior to Brooks' approach [11], robotics had been characterised by an approach of dividing up the central & peripheral systems of a robot into functional modules, operating in a sequential manner.

The alternative is decomposition by activity, that is, the desired external manifestation of the robot control systems. To this end, a number of "levels of competence" are defined. These levels are an informal specification of a desired class of behaviour for a robot given its environment. A higher level of competence implies a more specific class of behaviours.

The intention is that layers of a control system corresponding to each "level of competence" can be built and one can simply add a new layer to an existing set to move up to the next level of overall ability (how the robot "acts" when seen as a whole).

Brooks states that, in order to build systems based on activity decomposition so that they are truly robust, we must vigorously follow a careful methodology.

Firstly, anything constructed needs to be tested in the real world. A simplified "toy" world would make it all too easy to build a component of the overall system that relies on one of the properties of the "fake" environment. Depending on where in the hierarchy this simplification occurs the "error" may propagate to many other systems resulting in a final entity that is useless in the real world.

On a related note, Brooks argues that each layer needs to be tested in the real world as it is completed. Behaviours need to be observed and debugged otherwise errors become much harder to track down due to the large number of potential unchecked sources. Steels [12] argues that the development of these behaviour systems is very much an artform and, as such, a great deal of trial and error is to be expected. But he also suggests a set of design guidelines to help focus this creative process, the most relevant of which are reproduced here:

Guideline 1: Make behaviour systems as specific as possible. "Focused" behaviours are more efficient than "generic" ones.
Guideline 2: Simple mechanisms may give rise to complex behaviour. Make mechanisms as simple as possible and to rely on interactions between these mechanisms and the environment in order to get the desired behaviour.

Obviously applications other than behavioural systems exhibit emergent properties and Wooldridge et al present Gaia [13], a methodology for agent oriented analysis and design, for general usage in multiagent systems.

Gaia is a dual-phase methodology; comprising an analysis stage and a design phase. Both of which are summarised here:

Analysis phase:
1. Identify the "roles" in the system. These typically correspond to individuals, either within an organisation or independent; but may represent an entire collective.
2. For each role, identify and document the associated protocols. These are the interactions that occur in the system between the various roles.
3. Using the protocol model as a basis, elaborate the roles model.
4. Iterate stages 1-3.

Design phase:
1. Create an agent model: aggregate roles into agent types, and refine to form an agent type hierarchy; document the instances of each agent type using instance annotations. Agent types are comparable to objects, templates to be instantiated.
2. Develop a services model by examining activities, protocols, and other properties of the roles. These services are just functions that agents can perform.
3. Develop an acquaintance model from the interaction model and agent model. This is simply a graph of the communication links that exist between agents.

At first glance, this methodology may seem suited to our purposes but, as Potgieter & Bishop [7] argue, the Gaia methodology is not suitable for "complex adaptive" (emergent and self-organising) systems. It was intended for use in closed systems of distributed problem solvers, in which autonomous agents collaborate to achieve a global goal.

Potgieter and Bishop have defined BaBe (Bayesian Behaviour), a methodology that uses Bayesian behaviour networks to model and observe emergent behaviour. This consists of three phases; analysis, design and deployment.

Two models are defined during the analysis phase; a Roles Model – a description of each role in terms of responsibilities, permissions, interactions, and activities; and an Interaction Model – a description of each interaction and the agents involved.

During the design phase another three models are defined; an Agent Model – the agents composing the system; the Internal Behaviour Model – the behaviours of, and interactions between, agents; and the External Behaviour Model – interactions between agents and the environment.

These first two phases of the BaBe methodology are similar to the Gaia methodology. However, BaBe presents a final phase, Development, which contains an Emergence Model – the relationships between local behaviours and emergent global behaviour. This emergence model takes the form of heterarchies (relations of interdependence) of Bayesian behaviour networks.

The makers of the BaBe methodology put a lot of emphasis on the "adaptive" side of their system, utilising these behaviour networks to observe and learn from resulting behaviour and to modify the system based on this. Essentially this can be seen as an automated observer on a higher level of abstraction than the system itself and can thus be generally considered a viable alternative to this system we are proposing. It is our contention, however, that a system such as BaBe is unnecessary, possibly even ill suited, for an application such as an iSurface that may constantly be in a state of flux with regards to functionality and activity.

4 Deus Ex Machina Methodology

Now we finally present the initial Deus Ex Machine methodology. It should be stressed that the development of this system is an ongoing process of refinement. Its presentation at this early stage is to introduce the iSurface system to the wider community and to encourage discussion and feedback on various merits or flaws.

4.1 About the problem domain

Firstly we will re-examine the problem domain in some more detail; describing the iCells and their inner workings along with introducing the simulator system used to conduct experimentation.

An iSurface, as previously stated, is an irregular ad hoc network of homogenous sensing, acting and computing elements known as iCells.

Now, it should be stressed that iCells are only identical in terms of hardware (sensors, memory, CPU, etc.) and basic functionality. The real functionality for an iCell comes from the agents inhabiting it. An iCell will start out life as a blank slate with nothing but its basic functionality. Agents may be transmitted into the iCell and added to a collective of agents; each agent being executed by the iCell and capable of using the iCell's communication, sensing and acting capabilities as well as interacting with the other internal agents.

Without agents, the iCell is still capable of various tasks. The most important of these is the relaying of messages. An iCell is constantly listening for message input. Once a message is received, it is added to an internal message bank for examination by any resident agents before the iCell system gets a chance at it. The type of message is looked at, should it match an "Inc" or "Dec" keyword then the message is rebroadcast into the world with a "hop count" increased or decreased as appropriate. In addition to this, the iCell maintains a short-term memory of messages it has received. Should the iCell receive the same message again it will just be ignored. The memory is time limited and after a certain period will be erased. The other major system task is the reception and installation of agents as previously discussed.

At this point is it important to note that all communication is undirected; that is, there are no links between iCells (who, unless an agent adds the relevant functionality, don't even "know" other iCells exist), instead iCells just broadcast signals into the wider world and passively wait to see if they "hear anything". Agents are initially added to the system from without; one of the reasons for the name of the methodology.

4.2 Simulation of the system

For the purposes of experimentation, two simulators have been developed. One is an iCell simulator; an emulator that runs as a standalone program. The other is an iSurface simulator; a virtual "physical" universe for the iCells to be embodied within.

The iCell simulator has been created in Java allowing it a great deal of cross-platform compatibility (and opening up the prospects of use on devices close to the intended iCell rather than the conventional PCs currently used). As a precaution against too poor performance, the iCell simulator maintains a list of IP addresses of "nearby" (in the iSurface's virtual world) iCells and emulates broadcasting by only sending messages to these addresses. Agents are also written in Java, their bytecode can be passed between simulators and easily installed and executed.

The iSurface simulator is also a Java program. It provides a graphical representation of the virtual space with all active iCells displayed. These iCells are placeable in the "world" and one can apply sensor inputs (pressure, light, etc.) to the relevant simulators. In addition, the iSurface simulator provides an "agent loader", the mechanism for adding agents to the iCells by simply clicking the target.

4.3 Programming an iSurface

Having covered the basics of the system we can now discuss how we can give it useful functionality. We've established that an iSurface is a massively multi-agent system that has the potential to be in a constant state of flux with regards to functionality and activity and, factoring in failures and external influences from the environment, it can be labelled as a chaotic system. What do we do when things are too complicated to deal with? We try to simplify things.

We introduce the concept of a "feature"; an area of iSurface that is to perform a specific task. Instead of having an iSurface composed of thousands of iCells, we treat an iSurface as a collection of features existing in the Ether. These features can be seen as self-contained, self-interested entities who release information into the Ether for propagation around the "universe" (much like the old theories about light travelling via waves in the Ether like "sound" does through a fluid). Should a feature pluck a message from the Ether it may choose to act upon it.

Now, a feature consists of a number of iCells, each loaded with the necessary agents to produce the required functionality. The actual process by which the feature is created is irrelevant to this discussion; a user may "seed" a single cell and the feature grows from there, or a user may explicitly place a feature on the iSurface cell by cell, or a mixture of the two with a user specifying an outline and flood filling it; all that matters is that features are added by the user to the iSurface.

It is possible for features to overlap, the shared iCells possessing both sets of necessary agents. This shouldn't be a problem; as far as each feature is concerned it is the only thing in existence, floating alone in the Ether. This oft-mentioned Ether is in fact iCells using their base functionality of relaying messages.

This practice of simplification brings us back to our earlier mention of the observation of emergent behaviour being dependent on the level of abstraction we are viewing the system from. What the introduction of feature has done is produce another level of abstraction between iCell level and iSurface level (it can be argued that there is also an agent level below the iCell). Also recall our criteria for emergence; any complex system is emergent unless the macrobehaviour is simply a scaled-up version of the microbehaviour or if the interactions between agents are explicitly dictated by an external agent on a higher abstraction level.

This definition suggests that as long as we have two or more indirectly interacting features on an iSurface then the overall iSurface system is emergent, regardless of how these features are constructed (they may simply be a macrobehaviour which is a scaled-up version of the components' microbehaviour).

Of course, once you have emergent phenomena at one level you will have emergent phenomena at all higher levels (although not necessarily the same manifestation). Where you draw the base line on levels of abstraction to indicate atomicity is really where you define emergence in relation to. An atomic component, by definition, cannot be emergent. In this iSurface scenario, we define the agents and messages as atomic components.

Having argued the semantics we can discuss the guidelines for iSurface programming. It is always important to remember that an iSurface is a massively multi-agent system and is inherently unreliable. It would be impossible to program each iCell individually, especially as some may be faulty. Therefore a user is recommended to utilise self-organising principles in the creation of features. This inherent unreliableness means that one can never be sure about network structure and thus it is inadvisable to plan for a "perfect" feature. Instead, designing a system that will work just well enough in most situations if the preferred approach (the acronym of choice for this approach is ANADI, supposedly an engineering expression).

4.4 The Methodology

With these guiding principles in mind, it is time to discuss the methodology proper. It consists of three phases; two of which being software development and the third being deployment onto the iSurface. This is a very simple methodology, essentially based around decomposing functionality, but, because of the domain, this approach is ideal. Later in this paper is a worked example of this methodology in use.

The first phase is Application Design. This is essentially deciding upon what feature set we would like in our iSurface. The designer needs to specify what features are needed, their functionality, and their inputs and outputs into and from the Ether.

The second phase is recursive and takes place for each defined feature. This is feature design where the where the designer takes the functionality of the feature and defines agents that produce this functionality. Firstly, keeping in mind that iCells operate on a localised indirect communication basis, the designer must decide how to add the feature to the iSurface. It may be necessary to create sub-features whose macrobehaviour is scaled-up up microbehaviour. Again, inter-feature communication would be used. Next the designer needs to define the agents that will provide the necessary functionality. It is likely that these will be part of the sub-features used to generate the original feature. A designer can keep breaking features down into sub-features (repeating phase 2 for each, hence the label of recursive for this phase) until macrobehaviour identical to microbehaviour (which preferably should be kept as simple as possible).
By our previous definition, the moment two distinct "atomic" sub-features indirectly interact they form a complex system that exhibits emergent properties. This of course means that all higher levels of abstraction have emergent properties of some description.

The third and final phase is the deployment of the features in the iSurface. The user needs to place the sub-features of a feature onto the surface in the right order. In practice, this phase most resembles the use of a computer based paint package except that the tools and colours used are the developed agents. This is when the guidelines suggested earlier really come into play. Aiming for a particular network structure during design will cause problems when the user attempts to deploy the features. This is the "real world" test as described in methodologies for developing behavioural systems.

5 A Button Feature and Light Feature – a Worked Example

Here we perform a simple experiment for two features; a button & a light. We want the button to detect when it has been pressed and to alter state; i.e. if it were in an on state it needs to switch to off and vice versa. It also needs to release a message into the Ether stating that there's an on / off signal. The light needs to pluck messages from the Ether and alter its state to match what it finds; i.e. an on message will cause it to glow, and an off message will switch it off.

In both cases, state must be preserved across the features; all of the button must register on or off and all of the light must glow or be dark.

5.1 Phase 1 – Application Design

We have already covered pretty much what the overall application is expected to do. Here we define the features in an appropriate format.

Name: Button
Physical inputs: Pressure
Physical outputs: None
Message inputs: None
Message Outputs: ON, OFF (both have "Inc" keyword in order to be propagated through the Ether)
Functionality: When pressure is detected. The feature toggles its on / off state and releases the appropriate message output into the ether.

Name: Light
Physical inputs: None
Physical outputs: Light / no light (LED luminance)
Message inputs: ON, OFF
Message outputs: None
Functionality: When an ON or OFF message is received the feature the changes its status accordingly. In addition, the "hop counts" of the ON or OFF messages are output to demonstrate the indirect nature of the interaction.

It would be possible to represent these features in other formats but this will suffice for this example.

5.2 Phase 2 – Feature design

Now we recursively decompose these features in subject until the macrobehaviour of a sub-feature matches the microbehaviour of the agents necessary to perform the functionality of the sub-feature.
We will use a depth-first approach, concentrating first on the button and then on the light.

The button:
Firstly we need to decide how to add the button to the iSurface. For this example we will draw an outline for the button and then flood-fill it. This is actually two sub-features, the outline and flood-fill.
Luckily the flood-fill feature can act as the actual pressure sensitive component of the button and the outline can act as an "interface" to the Ether.
We therefore define our sub-features.

Name: Outline
Physical inputs: None
Physical outputs: None
Message inputs: Agent, Switch-on, switch-off
Message outputs: ON, OFF (both are "Inc" types)
Functionality: Outline agents constantly check the message bank of their host iCell. If the message is an agent of type "flood-fill" then it is deleted. If the message is "Switch-on" then it releases an ON message. If the message is "Switch-off" then it releases an OFF message.

Name: Flood-fill
Physical inputs: Pressure
Physical outputs: None
Message inputs: ON, OFF, Switch-on, Switch-off
Message outputs: Switch-on, Switch-off, Agent
Functionality: On start-up, flood-fill broadcasts itself as an Agent message. It should be noted that an iCell automatically ignores an incoming agent that it already has. The agent continuously checks its host's messages. An ON or OFF message is deleted (there's no point in transferring an Ether-bound ON / OFF signal across a button). If it sees a "Switch-on" message and its internal state is off then it toggles to on and broadcasts a "Switch-on" message. If it's a "Switch-off" message and the internal state is on, it toggles to off and broadcasts a "Switch-off" message. If it detects pressure it will toggle its state and broadcast the appropriate message.

The light:
We will base the light on the same ideas of the button, an outline being defined and a flood-fill being utilised. This time we want the outline to detect and delete the ON / OFF signals and broadcast a switch-on / off message. We also want to output the "hop count" of these messages before we delete them in order to prove that indirect interaction is occurring.

Name: L-Outline
Physical inputs: None
Physical outputs: None
Message inputs: ON, OFF, Agent
Message outputs: Switch-on, Switch-off
Functionality: Constantly checks the message bank of its host. If a message is an "Agent" of type of "L-Flood-fill" then is deleted. If a message is either ON / OFF then the "hop count" is reported, the message is deleted and the appropriate Switch-on / Switch-off message is broadcast.

Name: L-Flood-fill
Physical inputs: None
Physical outputs: Light
Message inputs: Switch-on, Switch-off
Message outputs: Agent, Switch-on, Switch-off
Functionality: On start-up, automatically broadcasts itself as an agent message. It continuously checks its host's message bank. If it finds a message of Switch-on / Switch-off at odds with its internal state it will change its state and broadcast the relevant message.

5.3 Phase 3 – Deployment

Having designed and implement the four agents (button flood-fill, light flood-fill, button outline, and light outline) it is time to deploy them onto the iSurface.

This experiment was performed on several configurations of iCell distribution. In each case a collection of iCells was outlined with the light outline agent and another collection defined by the button outline. Seeding an iCell in the centre of these areas with the correct flood-fill agent produced the desired behaviour.

Distribution: Random
Number of iCells: 200
Average "connections": 4 ("connections" being the number of iCells within broadcast range)
Details: The features were defined on opposite sides of the environment and each was composed of roughly 20 iCells. Applying pressure to the button triggered the message diffusion wave we were expecting. An alteration in the iCell simulator code changed the iCell's colour when it relayed a message. A spreading wave was evident by watching the iSurface simulator's graphical display. As expected, the light feature

illuminated and the lowest hop count reported by the light outline agents was 11; thus showing that the message had passed through 11 cells between the two features.

Distribution: Dumbbell, two clusters of 10 iCells connected by a line of 10 iCells
Number of iCells: 30
Average "connections": 3
Details: Each cluster of the dumbbell was defined as a feature (button and light) with the closest iCell to each end of the line being defined as an outline agent. As would be expected, the messages travelled down this line and thus the recorded hop count was 8.

Distribution: Two clusters of 30 iCells
Number of iCells: 60
Average "connections": 6
Details: Here we had two clusters of iCells, completely separated from each other. In one cluster a button was defined, the other received a light. Messages from the button only propagated through its cluster. The light never activated.

6 Conclusions

The three experiments are demonstrative of three possible configurations that an iSurface could be in. The first, the dumbbell, can be seen as explicitly providing a communications channel between the two features. This is a fair assessment as obviously there is no real likelihood that an ON / OFF message may not reach its intended recipient.

The second experiment, a randomly distributed collection of iCells, removed this explicit link between the features in favour of countless potential routes for a message to take. When messages were released into the Ether (the iCells without agents), they propagated outwards from their point of origin. The moment the leading edge of this diffusion wave touched the light feature it illuminated. Although this was the shortest path it was still indirect communication. Outline agents at the "rear" of the light registered hop counts of on average 15.

The final experiments, the clusters in the void, demonstrates that there needs to be a medium between interacting features through which information can propagate. "Direct" communication between the features would imply that this gap between the clusters would be no barrier. Instead no interaction took place, proving that there is no direct connection between the features.

As iCells can only broadcast message to their neighbours all communication is undirected and so all interactions may be seen as indirect. There is always a degree of uncertainty when programming an iCell as you will never know what it can communicate with. This domain is such that a relatively simple methodology, when used with the appropriate guidelines, can be used, successfully, to add functionality to an iSurface.

6.1 Future Work

As previously mentioned, this is ongoing research. The immediate next stage is to develop a measure of how suitable an iSurface is for a given application. Many factors affect how an application will function on an iSurface, too low an iCell concentration and features may be ill formed, likewise if the communication range is too high or low. "Pathways" in the Ether may not "link" features that are supposed to indirectly communicate because of a high failure rate of iCells. Measures that take account of these factors will allow a designer to adapt their systems accordingly.

References

1. Callaghan, V., Clarke, G., Colley, M., Hagras, H.: Embedding Intelligence, Research Issues for Ubiquitous Computing. (2001)
2. Intelligent Clouds. IST Proposal (2001)
3. Katznelson, J.: Notes on Amorphous Computing. From web.
4. Abelson, H., Allen, D., Coore, D., Hanson, C., Homsy, G., Knight, T., Nagpal, R., Ruach, E., Sussman, G., Weiss, R.: Amorphous Computing. From web.
5. Butera, W.: Programming a Paintable Computer. From web (2002)
6. Darley, V.: Emergent Phenomena and Complexity. From web (1994)
7. Potgieter, A., Bishop, J.: The Engineering of Emergence in Complex Adaptive Systems. University of Pretoria (2002)
8. Johnson, S.: Emergence: The Interconnected Lives of Ants, Brains, Cities and Software. Penguin (2001)
9. Ronald, E., Sipper, M., Capcarre, M.: Design, Observation, Surprise! A Test of Emergence. From web (1999)
10. Brooks, R.: Intelligence Without Representation. MIT (1987)
11. Brooks, R.: A Robust Layered Control System for a Mobile Robot. MIT AI Memo 864 (1985)
12. Steels, L.: The Artificial Roots of Artificial Intelligence. Artificial Life Journal (Vol1.1). MIT Press (1994)
13. Wooldridge, M., Jenning, N., Kinny, D.: The Gaia Methodology for Agent-oriented Design. From web (2000)