Self-Growing Applications from Abstract Architectures

An Application to Data-Mediation Systems

Ada Diaconescu
CNRS LTCI, Télécom ParisTech
Paris, France
ada.diaconescu @ telecom-paristech .fr

Philippe Lalanda
LIG laboratory, University of Grenoble
Grenoble, France
philippe.lalanda @ imag .fr

***Abstract*— Imagine a distributed mediation application consisting of hundreds of thousands of interconnected nodes, collecting data from millions of pervasive sensors, processing data and delivering it to a myriad of business services. This application takes the form of an acyclic, directed graph. Its shape must continually adapt in response to changes in sensor availability, network layout and business objectives. This involves dynamically adding, configuring, migrating and removing graph nodes. A centralised Observer/Controller, or Autonomic Manager (AM), that controls lifecycle operations for the entire graph would neither scale with the system's size and adaptation frequency, nor survive in unpredictable environments. This paper proposes a decentralised solution for enabling mediation applications to self-grow and to self-adapt their shapes and behaviours. In this approach, applications can autonomously grow into instances of a predefined, abstract architectural model and continually adapt to their execution conditions. A proof-of-concept prototype was developed using a Java-based, Service Oriented Component technology – iPOJO / OSGi. Experimental results from a Home Monitoring data-mediation scenario show the applicability and viability of our approach. We believe that the proposed framework will enable applications to autonomously grow and survive in volatile execution environments, over extended time periods.**

Keywords - self-growing applications; decentralised control and self-organisation; dynamic model interpretation; service-oriented components; autonomic life-cycle management.

I. INTRODUCTION

Imagine a distributed mediation application consisting of hundreds of thousands of interconnected nodes that collect data from millions of pervasive sensors, process the data at various abstraction levels and deliver it to a myriad of business services. The application takes the form of an acyclic, directed graph, which connects multiple data sources to multiple data sinks [Figure 1]. Each graph node can collect data from several sources (external sources or other nodes), process data (based on a specific algorithm) and deliver the result to several sinks (other nodes or external sinks). In order to ensure service availability, such mediation application must frequently change its shape and behaviour when: data sources dynamically appear, change location or disappear; the underlying distributed platform evolves; and business services progress. Application modifications include adding, migrating, updating and removing nodes, so as to follow changes in data sources and sinks, incoming loads, resource availability or data-processing requirements.

The *lifecycle management* operations required in these scenarios involve deploying, instantiating, configuring, migrating or removing software components (or services), for each node involved. Automating such operations can greatly improve the efficiency and dependability of system administration procedures [1]. Typically, e.g. [2] - [4], a central controller ensures the coordination of all management operations by determining, implementing and adapting global instantiation solutions for the entire distributed system. Nonetheless, as the managed application's scale and required adaptation frequency increase, the reactivity of such central controllers becomes progressively harder to maintain. The problem of having a single point of failure must equally be addressed for ensuring overall application survivability.

In contrast to such centralised, top-down approaches, a different research community addresses this problem from a decentralised, bottom-up perspective. Solutions in this domain rely on decentralised self-organisation and emergence principles. Typically, e.g. [5]-[10], independent processes act and interact with each other locally, based on simple programs and partial system views. No central controller or global runtime system view exists. Global behaviour or structure emerges from the local activities of such decentralised processes. The major difficulty in these approaches consists in guiding or *controlling* the decentralised process. This problem can be reduced to deriving the local behaviours and interactions that guarantee the emergence of desired behaviours or structures.

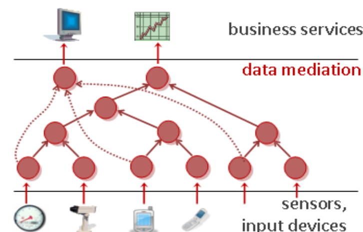

Figure 1: Mediation application

We address this challenge from a Software Engineering (SE) perspective, introducing a solution that borrows principles from both these domains. We propose CUBE, a decentralised framework for self-growing and self-adapting applications - i.e. lifecycle management. In this solution, identical agents - or *Autonomic Managers (AMs)*, replicate, specialise, self-organise and self-destroy so as to create and maintain a coherent application, adapted to varying contexts. To address the aforementioned control difficulty, we bring in an *abstract architectural model* for the overall application. This model is replicated into all AMs; AMs specialise in interpreting and expressing different model parts. Hence, the abstract model constitutes a global objective and each AM strives to attain a part of this objective. If local AM solutions can be composed linearly then the model provides the means of controlling the emerging application architecture.

In our solution, engineers merely specify the abstract model that defines the targeted application. This model defines architectural patterns, which authorise several degrees of variation. The model is then injected in an *initial set* of identical AMs (created by hand). Controlled by model definitions and constrained by actual runtime conditions, the initial AMs self-replicate, specialise, self-organise and self-destroy so as to produce an application variant that conforms to the engineer's objectives. As in traditional SE, resulting applications follow the "algorithmic division of labour" concept. Yet, these applications are opportunistically built and adapted via the dynamic composition of available services. A CUBE prototype was implemented using a Java-based Service Oriented Component technology – iPOJO / OSGi. Initial results from a Home Monitoring mediation scenario illustrate our solution's applicability and viability.

The most significant contribution of this paper consists in proposing a *decentralised*, *architecture-based* approach for self-growing, self-organising and self-adapting service-based applications. Our solution offers more flexibility and survivability than centralised, top-down approaches and better control and predictability than decentralised, bottom-up initiatives. While our initial study and prototype focus on data-mediation systems, the presented approach can most likely be generalised for other application domains.

II. SOLUTION OVERVIEW

A. *General Idea – a Living Systems Analogy*

For addressing the lifecycle management challenge, we searched for inspiration in existing complex adaptive systems - e.g. living systems. Indeed, most living systems are quite capable of autonomously growing and surviving in an impressive range of unexpected environments. In living systems, such as plants, an individual's development starts from a core element (e.g. a seed or a cell), containing a model - or *genome*, and a bootstrapping mechanism. The bootstrapping mechanism kicks-off the growth process by "reading" the genome and producing the necessary elements for duplicating the initial cell. Each cell contains a genome replica and the corresponding interpretation machinery. The process continues recursively and more cells are formed. In the primary cell cluster, all cells are identical. As the growth process progresses, cells start to differentiate into *specialised* cells, based on their position in the existing organism. Development unfolds concurrently from all existing cells, each *expressing* different genome parts. External conditions can influence growth and cause variation in individual sizes and shapes – *phenotypes*. At the same time, individuals developed from a certain genome conform more or less to the type, or *species*, that genome represents. On the resilience side, living entities can self-repair to a certain extent by growing replacements of most destroyed parts. These processes require no reliance on centralised control. Hence, living systems provide excellent inspiration for lifecycle management, as they can adapt within impressive ranges and recover from unpredictable conditions, while maintaining their conformance to predefined architectures.

We're merely looking into natural systems for inspiration, rather than attempting to faithfully replicate their intricate underlying processes. From this perspective, we have adopted the following nature-inspired concepts:

- *Avoid central control* in the instantiation and adaptation process; communication and coordination of decentralised parts (i.e. AMs) ensure overall coherence (e.g. reaction diffusion processes [6]);
- *Use a common abstract model*, replicated from AM to AM, to control the emerging result;
- *Use identical AM implementations* (one component); AM instances, and hence their roles and functions, differentiate based on dynamic configurations;
- *Create and specialise AMs progressively and concurrently*, following the shared abstract model;
- *Create and adapt partial instances*, representing different model parts, *depending on local runtime contexts* (e.g. available instances and resources);
- *Adapt partial instances* to changes in their local execution contexts and adjacent instance parts.

In this analogy, a CUBE abstract model represents the genome equivalent and the resulting application instance the phenotype counterpart. A CUBE AM, containing a full model replica and a partial application instance, resembles a specialised cell. In natural systems, a seed can be viewed as a device for setting in place the progressive self-organisation of nearby resources into a structured entity, representing an individual of a certain species. Similarly, a CUBE AM can be considered as a device for self-organising existing computing resources (e.g. various software services and hardware platforms) into an application of a predefined type.

B. *Main Architectural Principles*

CUBE's architecture is based on two main elements: a static, *abstract architectural model*; and a context-aware, decentralised model *interpreter* [Figure 2]. The *abstract architectural model* formally defines the application's

architectural constraints and possible variations. Specifically, architectural models define the Types, Interconnections and general Constraints that are common to all application instances (e.g. cardinalities, localisation or needed resources). They do *not* describe the runtime architecture of an executing application instance. This means that abstract models do *not* specify the exact service implementations to instantiate *nor* the exact runtime instances, interconnections or platform assignments. The *interpreter* dynamically decides upon these aspects instead. All application instances must comply with their abstract models. Yet, abstract model constructs enable several degrees of dynamic variation, via:

- Abstract Types, to be dynamically matched with service implementations of that Type or Sub-Type.
- Architectural Variations (branches), which are grouped under logical operators – e.g. *and, or*, *xor*. The branches to instantiate are selected at runtime.

Figure 3 depicts a graphical representation of a sample abstract model (a) and a few concrete instantiation solutions (b, c, d). It is important to note that service instances for a given Type (e.g. A, B, C or D) can be created from *different service implementations*, from various providers.

The context-aware *interpreter* receives an abstract model and produces compliant application instances, customised for the current execution context. The interpreter's instantiation logic is *decentralised*, consisting of multiple independent AMs. Each AM instantiates a *model fraction* of the entire model and joins the resulting *instance fraction* to instance fractions created by neighbouring AMs. AMs have identical implementations. Yet, each AM differentiates into a specific *AM Type* depending on the model fraction it must express – the *expressed model fraction*. Each AM must create, connect and adapt an instance fraction that matches the AM's expressed fraction. AM coordination relies on event-based communication, where AMs only react to events concerning their expressed fractions and neighbouring fractions. Based on these principles, the application instance progressively grows from adjacent fractions expressed by neighbouring AMs. Figure 4 indicates how multiple AMs produce a full application instance for the model exemplified in Figure 3-a.

C. Important Architectural Considerations

The presented solution raises several key concerns:

- Who, in turn, manages the AM lifecycles?
- How are architectural models split into fractions?
- How do independent AMs coordinate for ensuring long-term application coherence and adaptability?

In the remaining of this subsection we discuss several alternatives for addressing these issues.

1) AM Lifecycle Management

We propose that the AMs manage each other's lifecycles. Namely, each AM manages, and is being managed by, *neighbouring AMs*. To a certain AM, *neighbouring AMs* are those that manage model fractions that are adjacent to the AM's expressed fraction. Hence, each AM resolving a model fraction *finds*, *creates* or *repairs* AMs that resolve adjacent

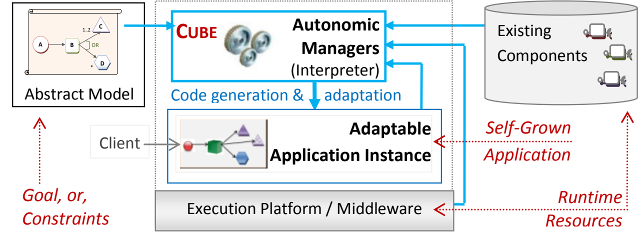

Figure 2: CUBE Architectural overview

fractions. Neighbouring AMs bind to each other and generate matching connections between their respective instance fractions. In this manner, the overall application instance and its management support grow progressively from one or more primary AMs. At runtime, the same logic is applied to detect and to repair neighbouring failed AMs or to re-instantiate application fractions in new execution contexts.

2) Model Fragmentation and Activation

Fragment activation is tightly related to the model fragmentation approach. We have so far identified four realistic options and implemented one of them in our prototype. First, one AM is instantiated on each distributed system machine. Fragmentation is decided at runtime, as follows. Upon creation, an AM starts to resolve the abstract model starting from a given point and until it encounters a model constraint (e.g. insufficient resources or instance localisation restrictions). At that point, the AM delegates the instantiation of the remaining, unresolved model to one or more neighbouring AMs (e.g. created on nearby machines). This process continues recursively until the entire model is covered. Second, an AM is instantiated for each *Component Type* in the model. An AM of a certain Type manages all Component Instances of that Type (e.g. as in Figure 4). In this case, fragments are implicitly defined by the model. Third, one AM is created for each *Component Instance*. Each

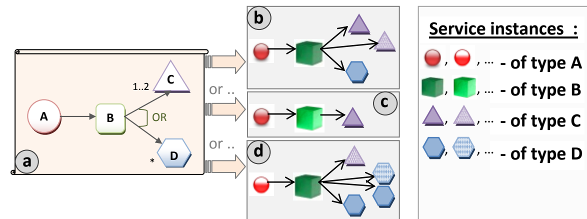

Figure 3: Abstract architectural model and instantiation examples

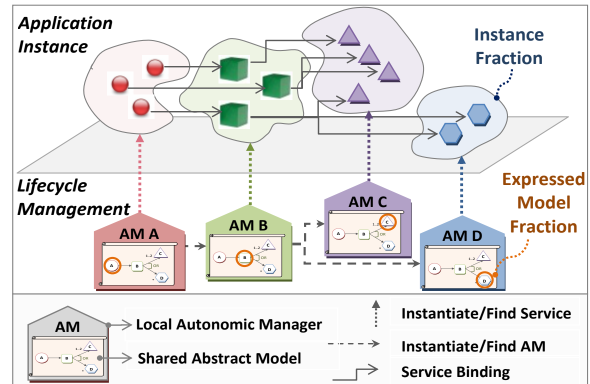

Figure 4: Decentralised instantiation process

AM has a Type and creates a single Component Instance of that Type. Like in the previous case, fragments are implicitly defined by the model. This third option was selected for our current prototype, due to its suitability for service-oriented technologies. Fourth, fragments are explicitly defined in the model and one AM is created for resolving each fragment. We believe that each of these approaches is valid and best suited for different system types. Various combinations can equally be foreseen for lifecycle management at different system scales. The adopted approach influences the AM coordination mechanisms, which were thus developed from the perspective of this particular choice (section V).

3) *AM Coordination for Global Coherence*

Global coordination is based on the AMs' local actions and intercommunication, guided by copies of the abstract architectural model. The core challenge lies in *synchronising* the parallel actions of independent AMs. Indeed, distributed AMs will have different clocks and messages between them may be lost, replicated or delayed. We identified and partially developed several coordination mechanisms (section V) for achieving the following goals:

- Avoid detrimental or needless instance redundancy, if AMs concurrently express the same fragment;
- Avoid isolated instance fragments that never join, as the application grows from multiple AMs;
- Avoid maintaining fragment instances that are no longer useful (i.e. garbage collection);
- Minimise the number of initial AMs that must be created by external means (i.e. bootstrapping);
- Minimise the occurrence and persistence of inconsistent application instantiation states;
- Avoid "growth flapping". Incompatible instance fragments can be partially destroyed and re-grown for better fitting. This process should be designed so as to ensure convergence within suitable delays.

III. Sample Application

A mediation system for Home Monitoring was selected for testing the proposed solution and the associated framework [Figure 5]. The sample system monitors the consumption of household resources, including electricity, gas and water. Collected data is processed for calculating different costs: house electricity, gas and water costs; region costs; and city costs. The corresponding Component Types include Specific Probes – collecting **e**lectricity, **w**ater and **g**as measurements and various Cost Calculators – computing consumption costs at the **h**ouse, **r**egion and **c**ity levels.

The selected example is a large-scale, distributed system with important dynamicity requirements. Households can frequently join and leave the system, requiring corresponding adjustments to the monitoring hierarchy. When home owners initially activate Specific Probes in their households, corresponding House Cost Calculators must be instantiated and connected to the appropriate Region Calculator. If a Region Cost Calculator is unavailable or overloaded, then it must be created or replicated and connected to the appropriate City Calculator. Such scenarios indicate the high administrative load required for maintaining this system coherent in the face of constant evolution. Our decentralised framework was designed to scale with the number of system nodes and with the frequency of local extensions or failures.

IV. Cube Framework Architecture

A. *Main Concepts and Functional Overview*

In CUBE, a model defines a set of application *Types*, *Associations* and *Constraints*. Context-aware interpreters, or AMs, decide *what* Types and Associations to instantiate and *when* (and *where*) to instantiate them. Currently, each AM features a configurable *type* parameter indicating its expressed Type. Each AM manages one Component Instance (or service) of its expressed Type. AM Types are assigned manually for the initial set of bootstrapping AMs. Grown AMs have their Types assigned by the AMs that create them.

The self-growing process proceeds as follows. Upon instantiation and validation, each AM receives its Type value and the architectural model. Based on these, an AM starts by positioning itself within the overall model - identifying the definition of its Type in the model. Once positioned, the AM creates a *local model* from the global architectural model. A local model is a star-shaped architectural fragment: its Centre consists of the AM's Type; its Ends consist of the Types Associated to the Central Type, as defined in the model. Next, the AM *resolves* its local model: acquires a service for the Central Type and Neighbour AMs for the End Types. Acquiring a service implies finding an existing service or instantiating one from a compatible Type. Finally, the AM binds to its Neighbour AMs and sets in place corresponding connections between their respective services. An AM becomes *valid* when it successfully resolves its local model. Once created, Neighbour AMs determine and resolve their own local models. In our proof-of-concept prototype we exclusively concentrated on the lifecycle management of AMs since we selected a one-to-one mapping between AMs and Component Instances. The creation and binding of actual services will be addressed in our future developments.

B. *Architectural Model Language*

We specified a model definition language for formalising abstract architectural models. We do not propose this language as a contribution, but as an enabling example for the proposed framework. Future work will study and adapt

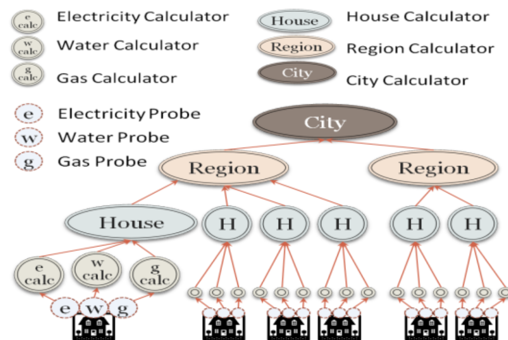

Figure 5. Sample home monitoring application

existing modelling languages (e.g. [1] or [12]). Our current model language is similar to those employed by Software Product Lines (e.g. [13]). It comprises two main elements – *Types* and *Associations*. *Types* contain Properties (e.g. id, cardinality), Constraints (e.g. suitable instance locations or resources) and Association References (i.e. relations with other Types; constrained by cardinality and organised into groups, with assigned operators - AND, OR and XOR).

Creation Policy is an essential model element specific to our approach. Defined for each Association Reference, it defines the management responsibility of AMs of the defined Type towards AMs of the Associated Types (i.e. growth direction). Creation Policies include: *find* - find an AM for the Associated Type; *find_or_create* - find an AM and if it does not exist then create one; or *nothing* - no responsibility. In Figure 4, AM A must find or create an AM of Type B, which must find or create AM C or AM D. Figure 6 exemplifies a Type definition for an Electricity Probe.

```
<!-- Electricity Probe -->
<type type="Electricity_Probe" id="elec_bp">
    <reference_associations>
        <ref_association id="elec-bp_elec-cost"
                cardinality="1"
                create_policy="find_or_create"/>
    </reference_associations>
</type>
```

Figure 6: Type definition example

```
<!-- Electricity Probe - Electricity Cost Calculator -->
<association id="elec-bp_elec-cost" >
   <end typeId="elec_bp" cardinality="1"/>
   <end typeId="elec_cost_calc" cardinality="1"/>
   <association-constraints>
                <endLocations>same</endLocations>
   </association-constraints>
</association>
```

Figure 7: Association definition example

Association definitions include two End Types and some additional Properties and Constraints (e.g. length, security or communication protocols). Figure 7 exemplifies an Association between the Electricity Probe and the Electricity Cost Calculator Types. Notably, the Location constraint states that both Association Ends must execute on the same machine (e.g. the home Gateway). In the prototype implementation, models are represented in xml format.

C. *Architectural Layers*

CUBE framework comprises several *AMs* and *Runtime Context Services* that execute on a specific technological *Platform*. The *Platform* must be in place on all stations on which the application is instantiated. The *Runtime Context Service* (or *Runtime*) provides relevant information on the current execution context. One Runtime service is available on each Platform, supplying AMs with an up-to-date view of the local context. This view includes information on the available AMs and Platform resources. For this purpose, each Runtime monitors its local Platform, intercepts local AM messages and exchanges information with Neighbour Runtimes (i.e. Runtimes of adjacent Platforms). Runtimes partially propagate local context states to Neighbour Runtimes in order to diffuse local events over adjacent areas. One or more AMs can be available on each Platform.

V. DECENTRALISED MECHANISMS

A. *Event Propagation and Scopes (Communication)*

In CUBE design, Runtimes intercept local events, store them and forward them to Neighbour Runtimes. This enables event diffusion over various *scopes*. Scopes can be defined as distances from a source (e.g. physical distance or maximum number of hops) or based on a location property (e.g. a network domain or an administrative area). Events carry scope-propagation data, which may be modified at each propagation step. Event Propagation is essential for AM coordination, providing the communication support upon which the other decentralised mechanisms are based. The current local prototype uses event-based communication.

B. *Instance Density Detection (Counting)*

Density represents the number of instances of a certain Type within a given scope. It is important for ensuring cardinality constraints. Currently, the one-to-one mapping between AMs and Component Instances facilitates this task. Upon creation, each AM signals its presence to the local Runtime. The signalled event contains a *marker* with the AM's Type and relevant properties (in an LDAP filter-like format). Additionally, the event contains the marker's validity period and scope. The local Runtime stores and diffuses the marker to Neighbouring Runtimes until the scope is covered. Each AM's *density* on a Platform is determined by the number of equivalent markers stored by the local Runtime. Runtimes periodically broadcast marker densities on the local and adjacent Platforms. AMs ask Runtimes for marker densities, which influence the AMs' subsequent behaviours. Our current prototype calculates Type densities in the local Runtime.

C. *Competition for Action (Leader Election)*

Competition is employed whenever a certain action must only be performed by a limited number of AMs within a certain scope – e.g. AM creation and destruction (subsection D). In this case, all AMs within that scope must compete for performing the action. The goal is to ensure the conformance of existing densities with the model's cardinality constraints. Currently, competition is based on a random count-down procedure. Competing AMs select a random number from which they count down. An AM wins if it finishes the countdown without being interrupted. When this happens, the winner broadcasts an inhibitor with a unique action marker. During countdown, all AMs listen for inhibitors carrying the marker of the action they compete for. AMs loose if their countdown is interrupted by such an inhibitor.

D. *Self-Replication and Self-Destruction*

These mechanisms self-regulate the density of AMs of a given Type. Self-Replication allows AMs to create more AMs of the same Type, while Self-Destruction enables excess AMs to self-destruct. AMs use the Compete for Action mechanism (subsection C) before self-replicating or self-destructing. This regulatory process is guided by instance density constraints (subsection B). Density constraints are defined in the architectural model; and

influenced by the current state of the growth process and by the execution context. Self-Replication and Self-Destruction can also be employed for adjusting Component Instance densities to fluctuating processing loads. Finally, if the Competition for Creation mechanism fails to prevent a Type density from crossing a threshold, then all AMs of that Type start competing and the losing AMs Self-Destruct.

E. Activity Desynchronization (Symmetry Breaking)

Desynchronization was introduced for preventing multiple AMs from simultaneously attempting to resolve local models and potentially cause resource consumption peaks or overflows. This situation may occur as applications are first instantiated, or as an important part of the application fails and must be re-constructed. In such cases, multiple AMs are concurrently trying to create or repair their local fragments, inducing important processing delays or event losses. This may impede the correct functioning of certain decentralised mechanisms. Desynchronization is based on selecting diverse waiting periods for delaying AM reactions to certain events – e.g. recreating failed AMs or regulating the Periodic Conformance Verification process (subsection F).

F. Periodic Conformance Verification

In CUBE, several mechanisms were set in place for coordinating independent system management processes (subsections B, C and D). Nonetheless, these mechanisms strongly rely on event communication, which may prove unreliable in most distributed scenarios (e.g. message loss or delay). For this reason, Periodic Conformance Verification is introduced *for preventing the occurrence, or persistence, of non-compliant instances*. This mechanism triggers the AMs' model resolution process at repeated, desynchronised intervals. AMs execute the same procedures for initial fragment instantiation, as for fragment repair or conformance verification; certainly, the required management actions will differ. This process ensures that accidental instantiation errors are detected and corrected over time.

VI. PROTOTYPE IMPLEMENTATION

A local framework prototype for the Home Monitoring application was implemented and validated, using a Java-based Service Oriented Component technology – iPOJO[1]/OSGi[2]. In addition to its inherent modularity and loose-coupling, iPOJO offers some essential functions for the runtime assembly and evolution of adaptable applications - e.g. dynamic component deployment, bundle dependency resolution, automatic service binding, service state change notifications, naming and directory service or event-based communication. The prototype Platform comprises a Java Virtual Machine, an OSGi implementation (i.e. Felix[3]) and the iPOJO runtime. The prototype was initially limited to a single Platform and one Runtime service. The Runtime and the AMs were implemented as iPOJO Services and include a core set of decentralised mechanisms: Competition for Action; Activity Desynchronization; and Density Detection. AM coordination and failure-detection rely on OSGi's event-based communication support. A graphical facility using Prefuse[4] toolkit displays the AMs' graph during runtime.

The current prototype focuses on creating and binding AMs of the correct Types. Creating the correct AM layer is equivalent to creating the correct application instance, since Component Instances and their bindings have a one-to-one mapping to AMs and their connections. The additional challenges ensued by actual component instantiation and binding remain the subject of our future research.

VII. TEST SCENARIOS AND RESULTS

CUBE prototype was tested for self-growing and self-repairing a local instance of the Home Monitoring application (section III). The architectural model employed included application Types (i.e. Electricity, Water and Gas Probes and various Cost Calculators), Associations and Constraints (e.g. cardinality, location and creation policies). The model imposes that each Probe (e.g. Electricity, Gas or Water Probe) be associated to a specific Cost Calculator, which must be situated on the same house gateway. Similarly, each specific Calculator must be associated to one Household Calculator, located at the same house address. Finally, Household Calculators must be associated to Region Calculators, which must be associated to a City Calculator. Household, Region and City Calculators must have the same city locations. Concerning creation policies, Probes must *find_or_create* matching Calculators, which *find_or_create* a Household Calculator. These must *find_or_create* a Region Calculator, which must *find_or_create* a City Calculator.

In this scenario, Electricity, Gas and Water Probe AMs were created via an external application. Starting from these initial AMs, the framework correctly created and bound the missing AMs so as to obtain a model-compliant mediation tree. Figure 8.a shows the mediation hierarchy obtained from the initial Water and Gas Probes, in the same household. In this case, the two Probes (step 1) determined their missing Cost Calculators and concomitantly created them (step 2). Next, the new Water and Gas Calculators concurrently detected the missing Household Calculator. The Competition for Creation mechanism ensured that only one of the two Calculators instantiated a Household Calculator (step 3). The losing Calculator waited, then found and connected to the created instance (step 4). In parallel, the Household Calculator created the Region Calculator (step 4), which created the City Calculator (step 5). In the displayed graphs, nodes represent AMs of different Types. Labels show the AMs' unique IDs, which contain the AM's Type and instance number suffix. Figure 8.b shows how the framework extended the initial hierarchy when two Electricity Probes were dynamically added - one to the existing and one to a different household. Subsequent scenarios tested CUBE's capability of repairing the mediation hierarchies – e.g. all Gas Calculators were removed from the existing tree and the framework reconstructed them.

[1] iPOJO Project: ipojo. org
[2] OSGi Alliance: osgi. org
[3] Apache Felix: felix. apache. org
[4] Prefuse visualisation toolkit: prefuse .org

We tested the prototype's capability to scale to the limits of the local machine resources (i.e. a mediation tree of 352 nodes: 150 Probes and 202 Calculators). These tests were important for identifying congestion-related challenges, such as the framework's behaviour in the presence of event losses. The Desynchronization mechanism was implemented to reduce the likelihood of such situations. The Periodic Conformance Verification will be introduced in future work to correct them. The goal of the presented scenarios was to show the successful coordination of decentralised processes based on the proposed architectural model and mechanisms. Performance evaluation was not a concern in these scenarios.

VIII. RELATED WORK

Contributions from two main domains are relevant to our proposal. First, Model-Driven Engineering (MDE) promotes models as domain-specific system abstractions from which executable computing programs can be generated. Such models have been increasingly introduced into the runtime environment to help autonomic management processes (e.g. [14] or [15]). Several model-based solutions propose application self-instantiation and self-repair (e.g. [1], [3] or [4]). Still, most of these approaches rely on concrete runtime models and centralised interpreting processes, which limit their flexibility and scalability. CUBE offers a decentralised solution, guided by static, abstract models; only model fragments must be locally maintained and interpreted during runtime. Similar to our proposal, the solution in [11] uses architectural constraints for self-organising components into applications that conform to a predefined architectural style. While this approach only focuses on the self-configuration of existing components, CUBE additionally manages the entire lifecycles of components and of AMs. Also, in [11], all managers receive and process all broadcasted events. CUBE AMs only process events relevant to their local models.

A second research domain related to our proposal comprises nature-inspired initiatives. Most relevant to our work, several projects adopt concepts from developmental biology, such as the genotype / phenotype paradigm, as an alternative means of Software Engineering complex adaptive systems e.g. [8]-[10]. CUBE provides a concrete framework that is compatible with these visions, while rendering self-development and self-organisation more controllable and predictable. In this context, several contributions define the software equivalent of the biological genotype as an unorganised collection of predefined behaviours. Identical components specialise by selecting the behaviours to activate and execute (e.g. [7]). CUBE views genotypes as architectural models, hence adding structural constraints to behavioural specialisation. This idea is compatible with the "Embryomorphic Engineering" direction promoted in [8], in which development of complex adaptable systems is achieved via the dynamic, decentralised interpretation of predefined meta-designs. Using the author's terminology, CUBE would represent an Intelligent Meta-Design (IMD) solution, if architectural models remained static and an Evolutionary Meta-Design (EMD) approach if architectural models were allowed to evolve (i.e. variation and selection).

Several interesting contributions have also been proposed in the multi-agent community. For example, in [9] or [10], an overall application plan is compiled into individual agent programs, in a way that ensures that correct global results emerge from the local agents' executions. Individual agent programs can be pre-differentiated for every agent [9] or identical for all agents [10]. These contributions focus on the self-organisation of already instantiated agents. In CUBE, AMs equally manage each others' lifecycles and the lifecycle of the application they create. Additionally, CUBE AMs are context-aware, and hence self-adaptable to their execution environments. Finally, CUBE separates the application-specific design, or model, from the AMs' interpretation logic. This separation is maintained during runtime, making the two parts independently reusable and evolvable.

In contrast to many biologically-inspired contributions (e.g. [8] or [10]), CUBE was designed for applications in which the application's physical shape (in Euclidian space) represents a minor concern, if at all; the application's behaviour or function is the main objective. Indeed, CUBE does provide support for constraints that can be location-related (e.g. minimum physical distance between deployed components). Still, CUBE's objective is to create "classic" software applications, consisting of specialised components connected in precise ways, in order to provide a well-defined function. In this context, CUBE's contribution consists in rendering the traditionally rigid design of such applications more flexible, adaptable and context-aware. CUBE can ensure overall application functionality in so far as application behaviour can be guaranteed by the application structure - i.e. correctly interconnected instances of well-defined component types. We believe that this is a reasonable assumption for applications in many specific domains, including the mediation domain targeted in our prototype.

IX. CONCLUSIONS AND FUTURE WORK

This paper presented CUBE, a decentralised solution for self-growing and adapting service-oriented mediation applications. Two key elements are at the core of our architecture: i) independent Autonomic Managers (AMs) that self-replicate, specialise for different application fragments, self-organise fragments into applications and self-destroy; and, ii) an abstract architectural model copied in all AMs, controlling their self-growth and self-organisation so as to ensure the emergence of core application properties, while enabling various degrees of application variation. An important characteristic of our design consists of the clear separation between the abstract architectural models; the decentralized, context-sensitive interpretation logic (AMs); and the produced application instances. These three elements can be separately evolved and reused. Most importantly, this enables the autonomic creation of application instances that are adapted to their execution contexts.

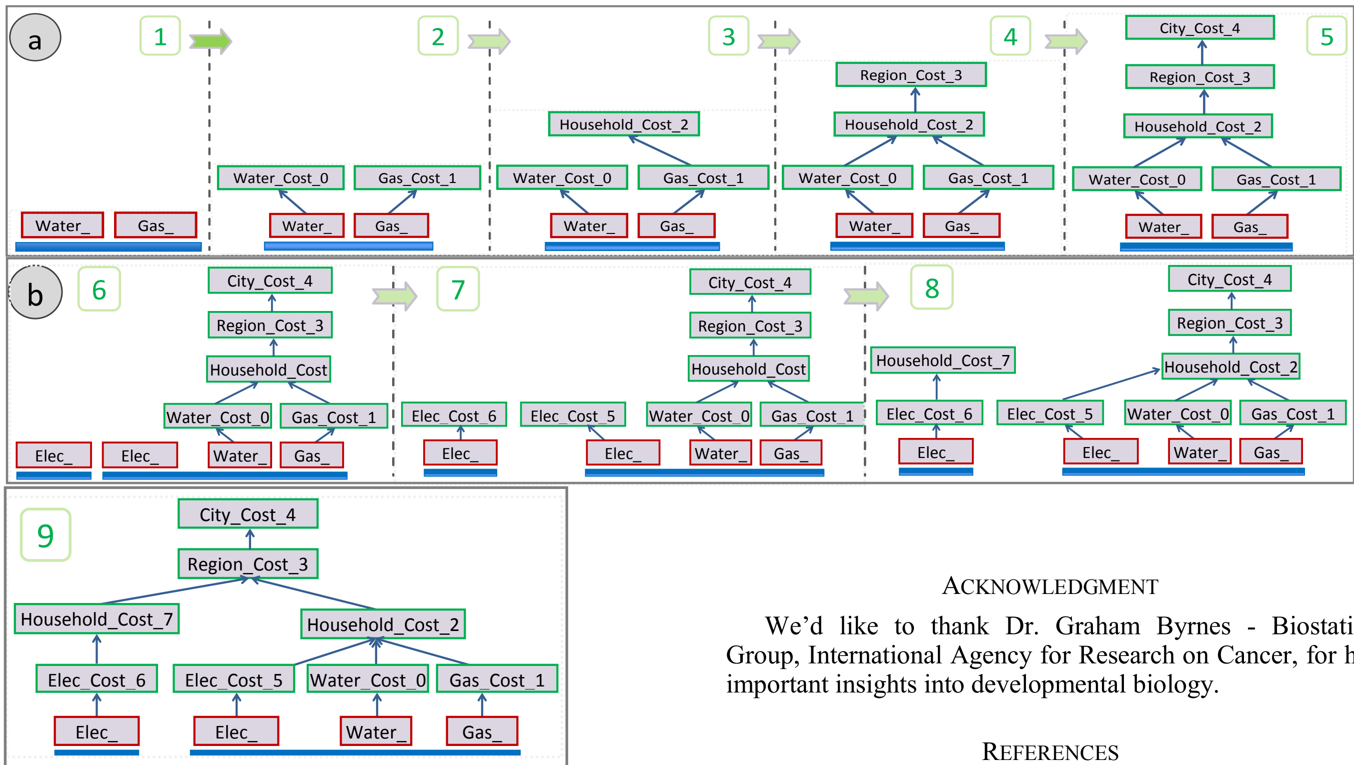

Figure 8. Home Monitoring hierarchy: a) initially grown from a Water and a Gas Probe, residing on the same household Gateway; b) extended after adding two Electricty Probes - one in the existing household and one in a different household; border: Red/Green for Seed/Grown AMs respectively.

This paper focused on presenting the main architectural elements and the core decentralised mechanisms identified so far for self-organisation and correct emergent behaviours. A framework prototype was implemented using iPOJO Service Oriented Components and successfully tested on a mediation application for Home Monitoring. The prototype showed the capacity of the proposed architecture to ensure the self-creation, extension and self-repair of coherent applications that meet a predefined goal. In the presented prototype, the mediation system's creation, evolution and long-term survivability were the key objectives. Performance will be considered as the framework evolves. The current prototype ensures the desired properties at a local level. While important challenges remain for extending the framework to distributed environments, we consider that our decentralised architecture and self-organisation mechanisms constitute a noteworthy contribution with respect to existing lifecycle management utilities. Further extensions will focus on: implementing and testing additional decentralised mechanisms; adding support for distributed Platforms; deploying, instantiating and binding Component Instances; and enriching the current model specification language (e.g. [1] or [12]). Finally, particular attention will be given to the study of existing nature-inspired design patterns (e.g. [6]). CUBE's long-term goal is to enable large-scale, distributed applications to autonomously grow and survive in volatile execution environments, over extended periods.

Acknowledgment

We'd like to thank Dr. Graham Byrnes - Biostatics Group, International Agency for Research on Cancer, for his important insights into developmental biology.

References

[1] J. Branke et al, "Organic Computing - Addressing Complexity by Controlled Self-Organisation", Intl Symp on Leveraging Applications of Formal Methods, Verification and Validation, pp 185-191, 2006

[2] P. Lalanda, L. Bellissard and R. Balter, "Asynchronous Mediation for Integrating Business and Operational Processes", IEEE Internet Computing, February 2006

[3] D. Garlan et al., "Rainbow: Architecture-based self-adaptation with reusable infrastructure" Computer, 37(10):46–54, 2004

[4] S. Sicard, F. Boyer and N. D. Palma, "Using components for architecture-based management: the self-repair case", International Conference on Software Engineering (ICSE), pp 101–110, 2008.

[5] S. Forrest, J. Balthrop, M. Glickman, D.Ackley, "Computation in the Wild", Internet as a Large-Scale Complex System, Oxfrd Press, 2002

[6] O. Babaoglu, et al., "Design Patterns from Biology for Distributed Computing", ACM TAAS, v.1, n.1, pp 26-66, 2006

[7] K. N. Lodding, "The Hitchhiker's Guide to Biomorphic Software", ACM Queue, v.2, n.4, pp 66-75, June 2004

[8] R. Doursat, "Organically grown architectures: creating decentralised, autonomous systems by embryomorphic engineering", Organic Computing, R. P. Würtz, ed., Springer-Verlag, pp 167-200, 2008

[9] R. Nagpal, "Programmable Self-Assembly Using Biologically-Inspired Multiagent Controll", International Conference on Autonomous Agents, Bologna, Italy, July 2002

[10] D.Coore, MIT PhD "BotanicalComputing: A developmental approach to generating interconnect topologies on amorphous computer" 1999

[11] I. Georgiadis, J. Magee, J. Kramer, "Self-organising software architectures for distributed systems", Workshop on Self-healing systems, pp 33-38, South Carolina, US, 2002

[12] OMG Specification, "Deployment and Configuration of Component-based Distributed Applications", v4.0.

[13] M. Matinlassi, "Comparison of Software Product Line Architecture Design Methods", ICSE, Edinburgh, Scotland, UK, 2004

[14] IEEE Computer, Special Issue on "Models @ Run.Time", Oct. 2009

[15] R. France and B. Rumpe, "Model-driven Development of Complex Software: A Research Roadmap", FSE, pp 37-54, USA, 2007.