Controlling self-organising software applications with archetypes

Bassem Debbabi, Ada Diaconescu, Philippe Lalanda

Controlling self-organising software applications with archetypes

Bassem Debbabi
LIG laboratory, University of Grenoble
Grenoble, France
Email: name.surname@imag.fr

Ada Diaconescu*
CNRS LTCI, Telecom ParisTech
Paris, France
Email: name.surname@telecom-paristech.fr

Philippe Lalanda
LIG laboratory, University of Grenoble
Grenoble, France
Email: name.surname@imag.fr

***Abstract*—Self-organisation is a promising solution for building complicated, large-scale software systems that must meet stringent adaptability and survivability requirements. At the same time, controlling self-organising software to ensure global system properties and functions is a difficult problem. This paper proposes a solution that uses architectural templates, or archetypes, replicated across a set of identical agents, and interpreted at runtime to control the agents' self-organising behaviour and results. The solution ensures, by construction, that any resulting software system meets a set of predefined goals, or constraints, while maintaining many of the self-organisation related advantages. A framework prototype was implemented and tested to show the viability of the proposed approach, in the context of a distributed data-mediation application.**

***Keywords*-self-organisation; self-growing software; architectural templates; autonomic lifecycle management.**

I. Introduction

Modern software systems seem to face two antagonistic requirements. To remain useful, they must constantly provide a predefined set of business functions and Quality of Service (QoS) properties. To manage their large-scales and adapt to ever changing conditions they must rely on decentralised processes that continuously reassemble their contents. Allowing a system to self-organise its internal composition during execution, while not impending on its core functionalities and properties, is a difficult task, at best. This paper presents an approach - called Cube[1] - that aims to simultaneously address these antagonistic requirements. The main idea relies on the use of a predefined architectural template, or *archetype*, which constrains the self-organising behaviour of a set of identical agents in order to control their produced results. In this solution, essential system goals are explicitly and formally specified in the archetype, which is then copied and distributed to all agents. Each agent is capable of reading, *interpreting* and *expressing* any part of the archetype, during runtime. This means that an agent is able to create and manage a software *application part* that conforms to the constraints defined in a corresponding *archetype part*. Creating an application part implies deploying, instantiating, configuring and interconnecting software components of various types while meeting the constraints defined in its archetype part. Each agent determines which archetype part to express depending on the application parts created by other agents. The final objective is to enable agents to self-organise so as to individually express different archetype parts and to interconnect the resulting application parts in order to collaboratively create a software application that matches the overall archetype. While all resulting applications match the archetype, each application may feature its own context-dependent specificities (e.g. concrete component implementations, number of instances and deployment platforms). This approach can be employed to autonomously manage the *lifecycle* of large-scale distributed applications, ensuring both their instantiation and subsequent adaptations.

The proposed solution raises three main types of difficulties. First, decentralised agent processes must be able to partition the archetype into complementary parts that cover the entire archetype. Second, agents must assign archetype parts among themselves so that each part, if it must be unique, is expressed by a single agent. Third, archetype constraints that span several archetype parts must be met via collaborations among the subset of agents involved in expressing those archetype parts. Several solutions are possible to address these difficulties - e.g., [1] [2] [3].

In the solution presented here, one agent is assigned to each execution platform in the system (Figure 1). The first two difficulties are simultaneously addressed as agents dynamically divide the archetype into parts. Each agent expresses a maximum of archetype components, starting from an externally-designated point and until local resolution capacities are reached. It then forwards the process to neighbouring agents, designating their archetype starting points. An agent's assigned archetype part is the part it manages to express. The process is kicked-off at system start-up and whenever the managed application changes. To address the third problem, coordination procedures are mediated via hierarchical, dynamically-designated $leader$ agents, which control access to properties that must hold within a certain application region, or $Scope$ (section II). This solution prevents multiple agents from simultaneously expressing the same archetype component, when the archetype forbids component replication. In case the number of agents to coordinate for ensuring non-local properties becomes extremely large, the hierarchical solution can be replaced with

* This work has been funded by MINALOGIC's MEDICAL project.
[1]Cube project: http://cube.imag.fr

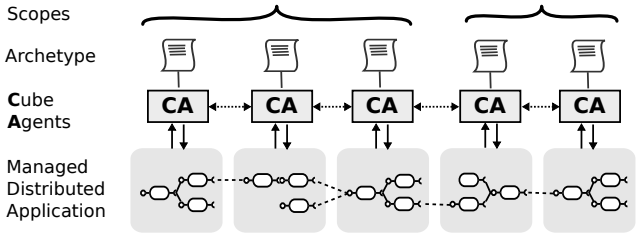

Figure 1. Cube agents managing distributed application

a completely decentralised algorithm (e.g. [4] or [5]).

Our previous work on this topic focused on highlighting the motivation and core principles of the Cube approach [1]; identifying archetype partitioning and agent assignment options, and exploring mutual agent creation and decentralised coordination [2]. An analogy between the proposed solution and morphogenesis - the biological organism development process - is briefly presented in [3]. These preliminary works allowed us to identify multiple strategic choices for implementing different Cube application parts. Yet, these parts proved difficult to alter or extend individually in order to create alternative variants for various scenarios.

To surpass this limitation, this paper proposes a *modular, adaptable and extensible Cube agent architecture and framework*. Choices made in the presented framework can be replaced with alternative approaches without disturbing the remaining agent functions. They currently include a hierarchical approach for Scope formation (previously defined dynamically based on event propagation distances [2]); an assignment policy associating one agent to each machine (previously assigning one agent to each component instance [2]); and a hybrid distributed coordination strategy (previously decentralised but confined to a single platform).

In addition, this paper identifies necessary archetype meta-types and defines a *formal extensible archetype language*. The language defines a core set of elements and supports various domain-specific extensions. Similarly, the presented framework provides a core set of functionalities and supports various *extensions* via well-defined plug-ins. A framework implementation was developed and evaluated for autonomously creating and managing a data-mediation application for home resource monitoring. Experimental results indicate the viability of the proposed approach and open several directions for future research.

II. Archetype-Controlled Self-Organisation

An *archetype* defines a system's architectural template, represented as a directed acyclic graph (DAG), with managed element *Types* as nodes (e.g. components or execution platforms) and *Constraints* between them as arcs (e.g. component interconnections or deployment preferences) (subsection IV-B). All agents receive an archetype copy. Agents are initially identical and can read, interpret and express any archetype part in order to produce a corresponding application part. At runtime, each agent *differentiates* and only focuses on expressing one archetype part. An agent's archetype expression starts from a pre-existing application component instance, created via an internal or external process (section IV-C). The agent first matches the component to existing archetype statement(s). The archetype point where the component is determined to fit represents the agent's starting point for archetype expression. Archetype expression basically involves finding or creating component instances of types defined in the archetype, and placing and connecting them as indicated in the archetype constraints. The agent progressively *resolves* interconnected constraints and generates a *growing application part*, until it reaches a constraint that it cannot resolve locally.

To forward the process, the agent must connect to agents that are specialised in expressing archetype elements that are bordering its own archetype part. If such specialised agents are already available, the agent must find and connect to them. It must then attempt to merge the application part it produced with these agents' respective application parts, by connecting component instances on the border of its part to component instances managed by neighbouring agents, following archetype constraints. If such agents do not yet exist, as is the case when the application is initially created, the agent must co-opt one or several non-specialised agents and point them to the archetype starting points that correspond to the researched elements. In either case, the newly co-opted agents continue the process in parallel, each one from its designated archetype position and until reaching its local limits, hence progressively defining and expressing additional archetype parts. The process stops when the entire archetype is covered and the corresponding application fully grown. Partial runtime failures trigger the agents that border missing application parts to re-organise, as in the initial procedure, filling-in detected gap(s), and regenerating a compliant solution (not yet implemented).

Agent synchronisation and coordination problems must be addressed to avoid overlapping archetype parts from being simultaneously assigned to multiple agents. Nonetheless, duplicated archetype parts only represent a problem when the repeated expression of such parts infringes on archetype constraints. For example, a constraint can impose that a single component instance of a certain type exists within a network domain. In this case, if an archetype part defining that component type is expressed more than once within that domain, the component will be instantiated several times and infringe the constraint. This problem is equivalent to the previously identified difficulty - ensuring a property that spans multiple archetype parts. To solve this difficulty the presented solution uses a hierarchical control design. It introduces a new archetype element - the *Scope* - representing an application deployment area within which a desired property must hold. For example, a Scope can represent an

administrative network domain, a geographical location, or a set of platforms sharing certain characteristics. A unique, property-specific Scope controller (or *Leader*) is elected among agents within each Scope. When having to resolve a non-local constraint within a Scope, agents must contact the Scope Leader to ensure the synchronisation and coordination of their parallel actions (e.g. the Leader indicates whether an instance of a certain type already exists).

Before the agent self-organisation process starts, an initialisation phase is executed to: assign each available system platform to one or several of the archetype-defined Scopes; to instantiate a Top Scope Leader at a location designated in the archetype; and, to designate one Scope Leader per Scope and register it with the Top Scope Leader.

III. Data-mediation system example

As an application example, we consider the autonomic lifecycle management of a distributed data-mediation system for monitoring the consumption of home resources, including electricity, gas and water. Generally, the purpose of data-mediation applications is to collect data from several *sources*, then transport and process the data so that it can be consumed by several *sinks*. Their architecture takes the form of a directed acyclic graph (DAG), where nodes are data-mediator components - or Mediators - (receiving, processing and forwarding data) and arcs are connections between Mediators (transmitting data). The use case considers multiple data sources representing resource monitoring probes and one sink representing a global cost calculator for consumed resources. The data-mediation application consists of a number of interconnected Mediators, each one processing data from sources or other Mediators to calculate consumption costs at various granularity levels, including home, city and country [2]. Cube's role is to create and maintain a data-mediation application that meets the architectural constraints specified in the designer's archetype. Experiments concentrate on the initial creation of the data-mediation application.

While the archetype of this sample system seems relatively simple (subsection V-A), actually managing the lifecycle of full-scale running applications conforming with archetype constraints can become highly difficult, risky and costly. Considering that millions of houses in a country may join the system, the number of mediators to be deployed on gateways and servers can similarly reach the order of millions. Moreover, whenever a house joins or leaves the system, corresponding data-mediation branches have to be set in place or removed from the overall system. As a simple example, consider that a new gas probe (GP) is added to one of the homes. Data provided by the new GP must be sent to a central house collector (HC), which merges data from all probes in that house. Implicitly, the GP mediator must be connected to a HC mediator. The HC instance must either exist already or be created on the gateway. Similarly, data from the HC must be transported to a city aggregator (CA), which merges data from a number of houses and sends it to city calculator (CC). The CC merges data from all CAs in a city and sends the results to a national aggregator (NA), which estimates the country's total consumption.

Considering for example that the mediation infrastructure is already set in place at the city and national level, let's see how Cube intervenes to automatically create the mediation branch for connecting the new GP to the existing mediation graph. The Cube agent managing the house gateway automatically detects the new GP instance and consults the archetype for any constraints on components of the GP type. In doing so it determines that GP must be connected to a HC within the same household. Hence, it searches for an existing HC on the local gateway and connects the GP to it. If an HC does not yet exist, Cube creates one and connects it to a CA that it must find in the same city; this is indicated in the HC's archetype constraints. During this process, the Cube agent on the gateway must contact the Cube agent on the server where the CA executes in order to connect the two mediators - HC and CA. At this point, the new branch is created and the resulting application shape conforms to the archetype. This was achieved without having to re-plan and re-implement the entire deployment schema of the overall mediation graph. In the following section we will reuse this simple scenario to exemplify the Cube archetype and resolver. Section V details the full example.

IV. Cube Framework

A. Cube Agent Internal Architecture

The internal architecture of a Cube agent follows a rather "classic" control loop structure (Figure 2). Technology-specific *Monitors* and *Executors* (Ms-Es) ensure data-collection and modification-reification from and into the managed application part, respectively. A *Resolver* component provides the control decision logic indirectly linking

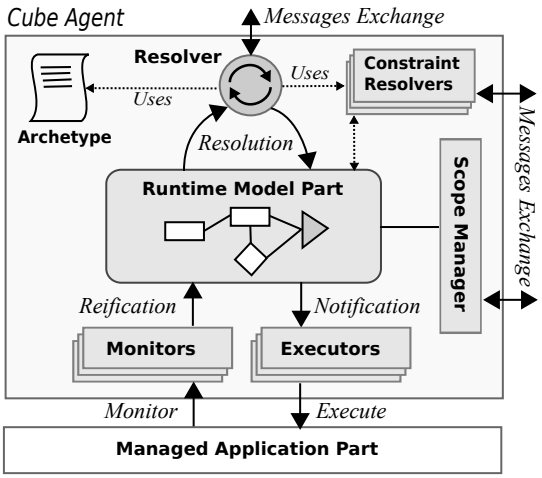

Figure 2. Cube agent internal architecture

data collection to application modification. The Resolver analyses the current state and context of the managed application part and plans changes for meeting management goals. Goals are expressed as a set of formally-defined constraints in the archetype. The Resolver actually consists of a core component (*Resolver* in Figure 2) and an extensible set of constraint-specific components (*Constraint Resolvers*). The core Resolver can read the archetype and create *Constraint Resolution Graphs* (subsection IV-C). Each Constraint Resolver (CR) is specialised in resolving one constraint type - e.g., one CR for component interconnection constraints and another one for component deployment constraints. To express an archetype part, the core Resolver creates the overall constraint graph of that part and forwards constraint resolution tasks to the corresponding CRs. This architecture allows extending both the archetype - via additional constraint types - and the agent Resolver that must express the archetype - via additional CRs. Similarly, specific Ms-Es can be introduced for each technology implementing the managed application part. Additional Monitors can be introduced to provide context-specific information.

Communication between the Ms-Es and the Resolver is mediated via a *Runtime Model Part*. This represents a local runtime view of the application part that the agent is managing. Model elements only provide information that is necessary for the agent Resolver to determine if they meet their archetype constraints. Ms-Es implement a *causal relation* [6] between the Managed Application Part and the Runtime Model Part. This means that any runtime modification in the Managed Application Part is reflected into the Runtime Model Part (via the Monitors) and any modification in the Runtime Model Part, once validated by the Resolver, is reflected into the Managed Application Part (via the Executors). In this context, for meeting the archetype goals, the agent Resolver aims to create and maintain a Runtime Model Part that meets the constraints specified in the expressed archetype part. Different CRs propose modifications to the Runtime Model Part in order to resolve different constraint types. The core Resolver arbitrates and solves potential conflicts among their propositions (not used in the presented use case). The Resolver validates the Runtime Model Part when it determines that it meets the constraints in the expressed archetype part. At that point, the concerned Executors are notified to reflect (or implement) the Runtime Model Part into the managed application part.

The borders of an agent's Runtime Model Part may represent links to remote elements in the Runtime Model Parts of neighbouring agents. This is the case whenever an interconnection constraint exists between components that were instantiated on different platforms, managed by different agents. When such cross-platform constraints exist, the concerned agents must collaborate and jointly solve the constraints. Consequently, before validating its own Runtime Model Part, an agent might have to wait until a collaborating agent confirms the resolution of a cross-platform constraint. Finally, at any one time, the complete Runtime Model of the overall application can be seen as partitioned into Runtime Model Parts managed by interconnected Cube agents.

B. Archetype Specification

Cube archetypes are specified in a descriptive, structured, mark-up meta-language (based on xml), which agent Resolvers can process at runtime. We defined a set of Cube-specific core meta-types[2] (xml tags), which can be extended as needed with domain-specific meta-types. The current prototype only uses the core meta-types, as follows.

A Cube archetype is divided into three main parts: *Types*, *Constraints* and *Global Configurations*. The first two parts define the administrative *goals* for the managed application's shape (e.g. interconnected component instances) and configuration (e.g. deployment settings). These goals represent the persistent objectives that Cube agents permanently strive to attain. The third archetype part provides invariant system configurations (exemplified in the use case).

Archetype *Types* define the application's managed elements, which can be of three core meta-types:

- *<component>*: a software component type;
- *<node>*: a type of deployment platform or physical device where component instances execute;
- *<scope>*: a deployment area, defined as a group of nodes of a certain type.

These core meta-types can be extended with domain-specific meta-types - e.g. *<data-mediator>* component type.

Archetype *Constraints* specify various restrictions, or conditions to be met, on the previously defined Types. They define the limits within which context-sensitive agents can opportunistically assemble available resources into various application instances, hence controlling essential application properties. Constraints can be *Unary* - involving a single managed element (e.g. limiting the number of input connections of a component instance - or simply component); or *Binary* - involving two elements (e.g. two components being connected). Notably, Constraints are directed, which implies that they only concern the Constraint's source managed element, while taking into account an existing destination element. For example, in Figure 4, the "connect" constraint directed from GP to HC implies that the Resolver must make sure that any GP instance is connected to a HC instance, but not vice-versa. This implies that GP is the only managed element directly concerned by this constraint.

Each Constraint type is tagged with a number of *markers* from a predefined set: *check [c]*, *find [f]* and *perform [p]*. Markers indicate the way in which the Resolver must use the Constraint during the resolution process and consequently the operations that the corresponding Constraint Resolvers

[2]A more thorough description of the archetype language is available from the project's Documentation and Catalogue sections - http://cube.imag.fr

must support (discussed in IV-C). Generally, [c] indicates that the constraint must be verified before the concerned managed element can be validated; [f] indicates that the constraint can be used for acquiring an instance of the constraint's concerned managed element; [p] indicates that the constraint may require modifications to the Runtime Model Part, for expressing solutions in the managed application. The Cube framework provides a predefined set of core Constraints, which can be applied to the aforementioned core Types and any of their extensions:

- <*connect* v1, v2>[c, f, p]: connect the two components indicated as values of variables v1 and v2;
- <*on-node* v1, v2>[c, f, p]: ensure that the component in v1's value is on the node in v2's value;
- <*in-components* v>[c]: ensure that the component in v has no more than a maximum number of input connections (where the max. is given as an attribute);
- <*out-components* v>[c]: ensure that the component in v has no more than a maximum number of outputs;
- <*in-scope* v1, v2>[c, f, p]: ensure that the node in v1's value is in the scope in v2's value;
- <*components-per-node* v>[c]: limit the maximum number of components that can execute on the node in v (where the type of components and the maximum value are given as attributes);
- <*components-per-scope* v>[c]: limit the number of components that execute in the scope in v;
- <*self-sizing* v>[c, f, p]: replicate components of v's type when their input connections are saturated;
- <*on-same-node* v1, v2>[c]: ensure that the two components in v1 and v2 are on the same node;
- <*in-same-scope* v1, v2>[c]: ensure that the two components in v1 and v2 are in the same scope;
- <*find-local* v>[f]: find on the local node an element of the type indicated by variable v;
- <*create-local* v>[f]: create component of v's type;
- <*find-scope* v>[f]: find a scope instance of v's type.

Additional Constraints can be defined as necessary for each targeted application. They must be matched by corresponding Constraint Resolver (CR) plug-ins.

Finally, the archetype's *Global Configurations* provide system property values shared by all Cube agents - e.g. the unique resource identifier (URI) of the Top Scope Leader. Figure 3 partially shows the archetype that defines the data-mediation example discussed in section III.

C. Cube Resolver

An agent Resolver must constantly ensure that the agent's model conforms to the archetype. To validate the model, the Resolver constructs and solves a directed *Constraints Resolution Graph* representing its archetype part, where the vertices are typed variables (corresponding to instances of archetype Types) and the arcs are constraints (corresponding to archetype Constraints). To solve the constraints graph

```
<archetype id="org.example.cube">
<types>
    <scope id="CITY"/>
    <node id="SERVER"/>
    <node id="GATEWAY"/>
    <component id="GP"/>
    <component id="HC"/>
    <component id="CA"/>
    <!-- more types -->
<types>
<constraints
    vars="gp:GP; hc:HC; ca:CA;
    g:GATEWAY; s:SERVER; c:CITY">
    <on-node v1="gp" v2="g"/>
    <connect v1="gp" v2="hc"/>
    <on-node v1="hc" v2="g" priority="1"/>
    <connect v1="hc" v2="ca" priority="3"/>
    <create-local v="hc" priority="2"/>
    <on-node v1="ca" v2="s"/>
    <in-scope v1="s" v2="c"/>
    <in-scope v1="g" v2="c"/>
    <find-scope v="c"/>
    <!-- more constraints -->
</constraints>
</archetype>
```

Figure 3. Partial archetype for house and city mediators

the Resolver uses a backtracking-based algorithm, trying to find an acceptable value for each graph variable. When a candidate value does not satisfy a variable's constraints, it is tagged as non viable and stored in the variable's history.

To solve each Constraint type, the Resolver calls a different specific *Constraint Resolver* (CR). Depending on each constraint's markers - [c], [f] and/or [p] - the associated CR provides the corresponding functions: *check()*, *find()*, *perform()* and/or *cancel()*. *Check()* verifies the constraint on a variable by assessing the conformity of the variable's value (true or false). *Find()* proposes valid values for a variable, either from its entire range, or considering given values for its related variables. *Perform()* updates the model to reify a constraint solution; *cancel()* annuls such operation.

The resolution process is triggered in three cases. First, an agent detects a model change (e.g. a new component, created manually). Second, the archetype imposes the existence of at least one instance of a Type; Cube agents must create it upon start-up. Third, a Cube agent asks another Cube agent to "find" a component instance. In all cases, once a managed element is found, the Resolver determines and validates its constraints. As a first step, the Resolver determines all the element's constraints marked as [c]. For each constraint, it calls *check()* on the corresponding CR. If the answer is True the element is valid. Otherwise, whether a new element value is obtained or the element remains invalid (i.e. no archetype solution). To check Binary constraints, a value must be found for the second element associated to the one being solved. Hence, as a second step, the Resolver determines all constraints marked as [f] for this second element, then calls *find()* on the associated CRs (in the order given by the constraints' priorities). As a third step, when a value for the second element is found, the Resolver calls *perform()* on the Binary constraint's CR, to update the model. At this point, the Resolver moves on to solving the second element and the process restarts recursively from this point. If the proposed element value cannot be validated, the Resolver calls *cancel()* and tries to "find" an alternative value. When the second element is valid the Binary constraint can also be validated along with the initial, concerned element.

Figure 4 shows how this resolution process works when a new GP instance is detected on a gateway. First, the

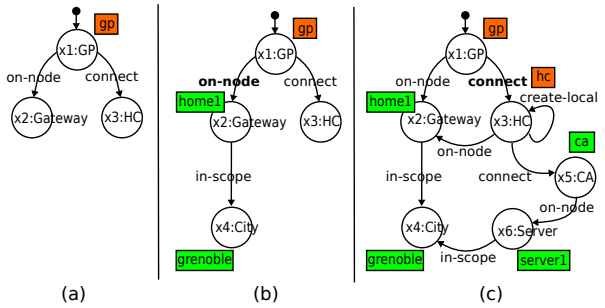

Figure 4. Example of Constraint Resolution Graph

Resolver creates a top variable ($x1$) of type GP and value gp (the detected instance) - Figure 4-a. To validate $x1$, the Resolver retrieves GP's [c] constraints from the archetype: <*on-node*>- gp must be on a Gateway Node; and, <*connect*>- gp must be connected to a HC component. To "check" the <*on-node*> constraint, a value for the related variable $x2$ of type Gateway must be found. Here, $x2$'s value is found directly from $x1$'s node attribute, available from the model (gp.node=home1). Next, the Resolver calls *perform()* on the "on-node" CR, adding gp to $home1$ in the model. Since $home1$ was changed, the Resolver must "check" its constraints. It retrieves its [c] constraints: <*in-scope*>- $home1$ must belong to a $City$ Scope. As before, the Resolver first tries to get a Scope value from $x2$'s attributes and it finds $Grenoble$, which is a valid value of type $City$. Hence, the "in-scope" CR returns True, then the "on-node" CR returns True, which validates this branch of GP's resolution graph (green values in Figure 4-b).

To resolve GP's <*connect*> constraint the Resolver must "find" a value for the $x3$ variable of type HC. Since it cannot find this value from $x1$'attributes, it determines HC's [f] constraints: <*on-node*>, <*create-local*> and <*connect*>, in this order of priority as specified in the archetype. Calling *find()* on the "on-node" CR returns no value, since a HC instance does not yet exist on the $home1$ gateway. Hence, the Resolver calls *find()* on the next CR - "create-local", which instantiates an HC component and returns its value. The Resolver assigns this value to $x3$ and calls *perform()* on the "connect" CR, between $x1$ and $x3$.

The Resolver must now "check" the new HC instance (Figure 4-c). Its "on-node" CR returns True, but its "connect" CR returns False (hc not yet connected to a CA instance). Hence, the Resolver first finds a value of type CA for $x5$. CA only has an <*on-node*> [f] constraint, placing it on a $Server$ Node. To find such Node ($x6$), the Resolver uses the <*in-scope*> [f] constraint associated to the $Server$ Type and obtains $server1$ in the scope $Grenoble$ of type $City$. The Cube agent on the gateway contacts the Cube agent on $server1$ for finding a CA instance. The remote agent constructs a local resolution graph with CA as its top element (not shown) and starts to solve it. It calls *find()* on the local <*on-node*> CR and obtains a ca value (since a CA instance already exists). This value is sent to the gateway agent, which assigns it to $x5$ and calls *perform()* to connect $x3$ to $x5$ (remote connection). It then rechecks $x5$, which remains valid, hence rendering the proposed HC instance valid and validating the new GP instance. The Resolver validates this solution locally and also with the other participating agents (the one controlling the ca instance on $server1$). When the model is validated, the Executors are notified to reify the solution into the managed application.

D. Essential Principles, Advantages and Limitations

Applications based on the presented approach conform to a fixed template (defined by an archetype) and support flexible template instantiations (context-sensitive solutions determined by agents). This approach offers a compromise between controlling essential application properties and enabling application survival and adaptation in changing runtime contexts. Self-organising Cube agents simultaneously and progressively find and create flexible application solutions, which allows the archetype expression process to scale. Agents do not have to determine *when* a complete solution is found before instantiating it. Also they will not attempt to find globally optimal solutions or to ensure 100% availability for managed systems. The main target is long-term survivability, adaptability and viability.

Using archetypes to express functional application goals renders Cube only applicable to cases where system architecture can guarantee system functionality. *Compositionality* is another essential assumption, asserting that if all archetype parts are correctly expressed and integrated then the resulting application is also correct (i.e. conforms to the archetype). This implies that a global solution can be obtained by composing local solutions that were developed with minimal mutual knowledge (i.e. local agent knowledge and Scope level properties). Both assumptions were considered reasonable for data-mediation systems considered so far.

With respect to the *decidability* of the constraint resolution process, the core constraints and resolvers provided so far are decidable if the number of available solutions for each constraint is finite (e.g. a limited number of implementations can be found for a component type). The resolution process sequentially addresses each constraint that applies to a managed element, using a backtracking process to progressively build and verify the solutions' tree. Each set of constraints is decidable if its solution tree is finite. For each managed element being expressed, once a solution that meets all its constraints is found, the solution is set in place and not rolled-back to accommodate other element's expression. This "greedy" approach ensures the convergence of the archetype resolution process, as the application grows monotonically towards a complete solution. Regarding the archetype's *expressiveness*, the extensibility of the provided language and associated resolver should ensure support

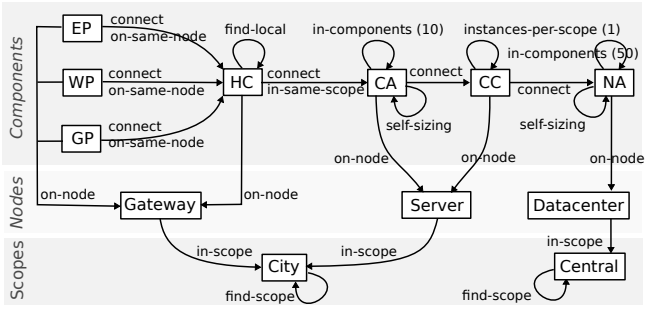

Figure 5. Complete constraints graph for use case archetype

for a large spectrum of domain-specific archetypes. The applicability of the proposed "greedy" resolution technique will have to be analysed case by case for each extension. Similarly, the *stability* of the resulting application will have to be studied depending on the context-sensitive adaptation of included Constraint Resolvers - this is a "classic" problem in autonomic and control systems. Supported *uncertainty* sources include the discovery of concrete Type instances that were not known at system design time - e.g. new component implementations or machines; dynamic communication networks impacting discovery; and evolving data flow sources.

V. Use Case

A. Home Resource Monitoring Archetype

Figure 5 provides a graphical view of the complete use case archetype. The archetype specifies two Scopes: $City$ - regrouping all Nodes in a city; and $Central$ - regrouping all Nodes of the national data centre. It also specifies three Node Types - $Gateway$, $Server$ and $Datacenter$ - and seven Component Types - Gas Probe (GP), Water Probe (WP), Electricity Probe (EP), House Cost Calculator (HC), City Aggregator (CA), City Calculator (CC) and National Aggregator (NA). Different instances of the defined Types can exist during runtime (e.g. $Paris$, $Grenoble$ and $Lyon$ are instances of the $City$ Scope Type).

Next, the archetype specifies various Constraints on the defined Types. It specifies the way Components are interconnected (<*connect*>) and assigned to Nodes (<*on-node*>and <*on-same-node*>) and to Scopes (<*in-same-scope*>). It defines the Node's inclusion into Scopes (<*in-scope*>) and the Scope's management strategy (<*find-scope*>). Notably, it imposes that a single CC instance can exist in a city (<*components-per-scope*>). For performance considerations, it limits to 10 the maximum number of inputs of CA instances (<*in-components max=10*>). Finally, for CA and NA types, it indicates that at least one non-saturated instance must be available at any one time (<*self-sizing*>).

B. Self-growing data-mediation chains

When Cube agents equipped with archetype copies are launched on the system's platforms, they first self-organise with respect to the archetype-specified Scopes. Scope Leaders are automatically elected and Node groups are formed within each Scope. Once the Scope infrastructure is set in place, Cube agents start creating local Mediator components and cooperating among themselves to construct distributed mediation chains. Let us examine this process considering the use case archetype (constraints graph in Figure 5).

In the Datacenter, the local Cube agent spontaneously creates the NA Mediator upon its initiation, because of the *self-size* archetype constraint specified on the NA Type. This constraint indicates that at least one accessible component instance of the concerned Type must be available at any one time. To validate this constraint, Cube agents create a new component instance when none can be found or whenever the existing ones are saturated. The same applies for CA components on Server machines, meaning that at least one CA instance is created when the Cube system starts. When CA is instantiated, a CC component must also be acquired - found or created - within the same Scope and connected to the CA instance. As a single CC instance can exist within any $City$ Scope ("instances-per-scope(1)" constraint on CC), Cube agents must ask the Scope Leader of their $City$ Scope for permission before creating a CC instance. If a CC already exists (i.e. created by another agent), the Scope Leader returns its reference and the demanding agent connects its CA instance to it.

Relying on this initial setting, when probes of type EP, GP or WP are installed in a home, Cube creates the corresponding mediation branches to connect them to the existing mediation graph, as illustrated in previous sections.

VI. Implementation and Experimental Results

The presented Cube prototype is implemented using Apache Felix iPOJO[3] technology, which runs on an OSGi[4] platform. IPOJO relies on a service-oriented component model where component dependencies are expressed as service requirements, which are dynamically provided by other components. Component implementations are packaged as *bundles* and can be hot-deployed on OSGi Service Platforms. Bundles can be deployed from local sites (e.g. file system) or from a remote repository (e.g. OSGi Bundle Repository). The Cube prototype capitalises on the dynamic capabilities of the iPOJO/OSGi service architecture to implement the different Cube agent modules. Remote repository and hot-deployment support allows Cube agents to dynamically find Component Type implementations and deploy them onto their local platforms. The service-oriented model enables Cube agents to maintain partially instantiated applications, where some application services can remain *unresolved* (rather than throwing exceptions) until the required services become available. Finally, Cube agents can

[3]http://felix.apache.org/site/apache-felix-ipojo.html
[4]http://www.osgi.org

feature a dynamically extensible architecture, where specific components (such as Constraint Resolvers or Ms-Es) can be deployed, instantiated and plugged-into agents at runtime, as needed to resolve various archetype parts. This enables Cube agent instances to be individually customised, at runtime, depending on their actual specialisation (expressed archetype parts). This helps minimise individual Cube agent overheads.

The presented prototype implements Cube's core framework - i.e., Resolver and Runtime Model container - and a set of specific extensions for enabling Cube to express the use case archetype. Extensions mainly include the necessary Constraint Resolvers - e.g. "on-node", "connect" and so on. Support for Cube agent communication is also implemented as an internal iPOJO component, currently based on TCP/IP sockets and easily replaceable as needed.

An initial set of experiments was carried-out in the described use case. Their main purpose was to validate the prototype's functionality in a distributed platform with respect to the archetype constraints identified so far. As the archetype and resolution process were the main experimental targets, mediator components were not actually instantiated - achieving conforming model parts for each agent represented the equivalent result. Initial performance measurements taken in this context provide an indication of the order of magnitude of delays to be expected from an agent's resolution process. Most importantly, they indicate the way in which such delays will depend on the archetype part sizes and the agent collaboration involved. Most definitely, large-scale testing on a variety of scenarios and platforms is required in future work to provide a comprehensive performance evaluation of the Cube approach. Here, we mainly discuss which functional parts of the Cube resolution process are most likely to introduce the delays. At the same time, since agents work in parallel on separate platforms, we estimate that delay characteristics will remain similar to the ones presented here even as the number of agents increases considerably. Certainly, delays may increase if the number of agents that must coordinate their actions increases.

Table I
USE CASE SCOPES AND NODES INSTANCES

Scope Type	Scope Instance	Node Type	Node Instance
Central	central1	Datacenter	datacenter
City	Paris	Server	P-server1..3
		Gateway	P-home1..5
	Grenoble	Server	G-server1
		Gateway	G-home1..3

Cube prototype was tested using the configuration depicted in Table I. This configuration defines the available instances corresponding to the archetype-defined Scope and Node Types. For example, we defined one instance of the "Central" Scope Type - $central1$ and two instances of the "City" Scope Type - $Paris$ and $Grenoble$. Regarding Node instances, we provided one platform of "Datacenter" Type for the "Central" Scope instance and so on.

The actual testing platform consisted of three PCs connected via a local area network (LAN). The first PC represented the Datacentre Node (Pentium 4 2.0 GHz, 1 Gb RAM). The second PC (Core Duo 1.83 GHz,3 Gb RAM) hosted all Node instances of the Server Type ($P-server1$ to $P-server3$ and $G-server1$), each one running in a separate OSGi instance (and process) to simulate different machines. Finally, the third PC (Core 2 Duo 3.06 GHz, 4 Gb RAM) hosted all Node instances of the Gateway Type ($P-home1$ to $P-home5$ and $G-home1$ to $G-home3$), running in separate OSGi instances. All OSGi instances ran on an OpenJDK 1.6 JVM executing on a Linux Ubuntu OS.

Figure 6 depicts the average times (over 10 runs) that Cube agents on these platforms took to initially self-organise into Scopes and then to assign and resolve their archetype parts. Measured execution times only represent the time needed for agents to determine and resolve their local constraint graphs. They do not include the technology-specific delays for the actual creation and interconnection of Mediator instances once their models are validated.

Let us analyse the performance of different agents depending on their hosting Nodes and expressed archetype parts. With respect to initial Scope constitution, we notice that agents join their Scopes within a rather small lapse of time (less than 20 [ms]). This time will essentially depend on the communication delays between the agent and the Top Scope Leader, as well as on the Top Scope Leader's load at the time the agent contacts it. The archetype partition and resolution times for different Cube agents was also quite small (less than 200 [ms]) and will also critically depend on communication delays with Scope Leaders and other agents. Most notably, an agent's resolution delay will increase if it depends on the resolution process of another agent, as it must wait until this other agent validates its own local model.

Average resolution times were approximately the same across agents running on home gateways, since they all

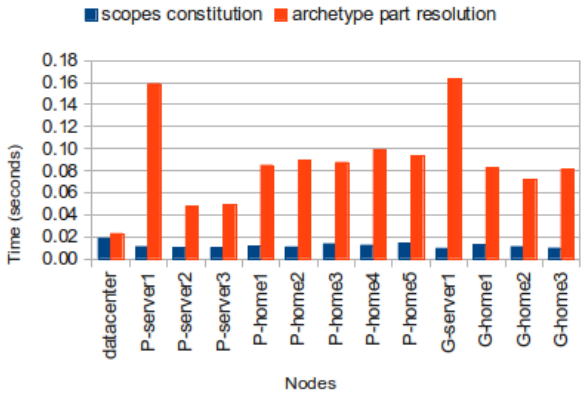

Figure 6. Average times of scope and archetype part resolution

performed similar tasks. Considering the servers, the average time on $P-server1$ is greater than that of similar servers in the $Paris$ Scope. As this server was always started first, it had to create the CC component that must be unique in the $Paris$ Scope and connect it to the NA component managed by the datacentre agent. The other $Paris$ servers simply connected their CA components to the existing CC. A similar situation can be observed for $G-server1$, which is the only one in the $Grenoble$ Scope. Finally, the resolution time is the smallest for the $datacenter$ agent, since it must only create the local NA instance.

VII. Related work

An exhaustive picture of related work is difficult to provide, since the presented approach finds itself at the intersection of several research fields and sub-fields. We focus on positioning Cube's autonomic lifecycle management approach considering two key engineering currents: i) "traditional" model-based approaches (top-down), including Model Driven Engineering, model-based self-deployment and self-management; and, ii) relatively "recent", nature-inspired approaches (bottom-up), including self-organising systems, emergent control or embryomorphic engineering. Cube positions itself in-between these poles, using archetypes to control the results of self-organising processes.

Model-driven top-down solutions typically define an architectural model of the targeted application and employ an automatic interpreter for generating, deploying, instantiating, (re)configuring and/or repairing a distributed application. In this context, architectural models can be *concrete* - defining the precise application to instantiate and manage (e.g. [7] and [8]); or *abstract* - only indicating the application types, type dependencies and or general constraints (e.g. [9], [10]).

Automatic model interpreters can work whether *off-line* - for initially creating model-compliant application instances (e.g. [8]), or *online* - for providing dynamic management functions (e.g. [7], [9] and [10]). Cube uses abstract architectural models (via the archetype) and online interpretation and expression (via the agents). An important limitation in most model-oriented approaches stems from *centralising* both the model interpretation processes and the Runtime Model of the resulting application (e.g., [7] and [9]). This limits the scalability of the targeted application both in terms of the number of managed components and of the frequency of required adaptation operations. Cube avoids this difficulty by employing self-organising agents as interpreters, each one only expressing limited archetype parts and maintaining local Runtime Model Parts of application parts.

Seen from the "bottom-up" perspective, this approach can be viewed as a means of introducing better control in self-organising and/or emergent systems. Similarly, [11] proposes to have decentralised processes guided by either a shared template - i.e. abstract architectural model, or by a shared recipe - i.e. set of rules. Cube perfectly fits this category and features important resemblances with existing solutions that adopted similar approaches - e.g. [10] or [12]. In [10], abstract architectural models are employed to guide the self-organisation of pre-existing software services into a global application solution. This approach adopts aggregate gossiping to exchange and merge partial solution configurations among participating processes. When a complete, model-compliant solution is reached it is used to guide the actual application self-assembly process. Cube does not try to explicitly find a global solution before setting it into practice. Instead, each agent creates a partial solution and instantiates it as soon as it can be validated. Application parts can be created (or removed) and plugged-into (or out of) the existing application later on, without having to restart the entire self-assembly process. In the Organic Computing context, [12] introduces the Restore Invariant Approach (RIA), in which a system's state is being continually verified against a predefined invariant (i.e. constraint or 'behaviour corridor') and reconfigured whenever it deviates from its viability space. Cube imposes archetypes as a particular kind of invariant and provides a reusable and extensible definition language and framework for implementing this solution.

Cube resembles certain clustering-based approaches for decentralised self-management (e.g. [13]), which employ task fragmentation and coordination among parallel processes. In the exemplified load-balancing solution in [13] local results combine straight-forwardly into a global solution. As this cannot be assumed in the contexts we target, Cube provides specific agent coordination techniques for obtaining conforming global solutions from multiple local parts.

In the context of "bottom-up", Nature-inspired research, 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. [14] to [18]. Cube provides a concrete framework that is compatible with these visions, where an archetype can be seen as a genotype, and agent-created application instances as phenotypes. Most existing approaches use predefined behaviours, or rules, to define individual genotypes [14], [15] or [16]. The parallel execution of multiple agents implementing such rules leads to the emergence of a desirable system behaviour and/or structure. In some cases, alternative behaviours are available and can be dynamically selected in each agent [17]. In contrast to such rule-based approaches, Cube proposes a goal-oriented solution - an archetype represents a goal that Cube agents must collectively attain. This approach simplifies system development, as it allows developers to specify the targeted end-result system (the *what*), rather than the means of achieving that result (the *how*). Additionally, Cube's dynamic archetype interpretation facilitates the creation of adaptable, context-aware applications, where the execution environment and the existing application parts influence archetype expression.

Most similar to Cube, [18] proposes a morphogenetic en-

gineering approach for self-growing robots from functional blueprints. While the idea is similar, Cube targets software applications where the actual physical shape in Euclidian space is often of little importance. Additionally, software contexts raise adaptation challenges that are more diverse than the element growth and shrinkage in robotic systems. Finally, Cube must handle system growth as a continuous process rather than as a single initial event.

VIII. Conclusion and Future Work

This paper presented an approach for instantiating and managing distributed software applications via a set of self-organising agents, controlled by a replicated archetype. The archetype represents the generic features that will inevitably occur in all instantiated applications. Each application instance can be unique with respect to its concrete component implementations, number of instances and deployment on execution platforms. This solution combines the control capabilities of "traditional" Software Engineering methods for ensuring core system properties (i.e. what designers can know at system design time), with the flexibility of self-organizing methods for ensuring system survivability and self-adaption (i.e. what designers cannot predict and must allow agents to decide at runtime). This approach provides a possible solution for controlling self-organising systems.

To help implement this approach, we provided a reusable and extensible archetype definition language and resolution framework. The current design combines decentralised agent collaboration (whenever possible to ensure local properties) with property-specific hierarchical control (to ensure properties defined over larger Scopes). A framework prototype was implemented using iPOJO service-oriented component technology and tested to create a distributed data-mediation application for resource home monitoring. Initial results indicate the functional viability of the proposed solution.

Future work will concentrate on extending experiments to include system self-repair scenarios and evaluate performance in large-scale system contexts. In parallel, we also intend to study existing constraint-resolution engines and evaluate their applicability within our framework. Finally, we are interested in exploring alternative solutions to global agent coordination, possibly replacing Scope-based control hierarchies with completely decentralised protocols.

References

[1] A. Diaconescu and P. Lalanda, "A decentralized, architecture-based framework for self-growing applications," in *International conference on Autonomic computing ICAC*, 2009.

[2] A. Diaconescu and P. Lalanda, "Self-growing applications from abstract architectures an application to data-mediation systems," in *IEEE Workshop on Organic Computing*, 2011.

[3] A. Diaconescu, D. Bassem, and P. Lalanda, "Self-growing software from architectural blueprints," in *Morphogenetic Engineering Workshop MEW*, 2011.

[4] R. J. Anthony, "Emergence: A paradigm for robust and scalable distributed applications," in *International Conference on Autonomic Computing ICAC*, 2004.

[5] M. Jelasity, A. Montresor, and O. Babaoglu, "Gossip-based aggregation in large dynamic networks," *ACM Trans. Comput. Syst.*, vol. 23, 2005.

[6] H. Song, G. Huang, F. Chauvel, Y. Xiong, Z. Hu, Y. Sun, and H. Mei, "Supporting runtime software architecture: A bidirectional-transformation-based approach," *J. Syst. Softw.*, vol. 84, no. 5, 2011.

[7] S. wen Cheng, A. cheng Huang, D. Garlan, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *IEEE Computer*, vol. 37, 2004.

[8] OMG, "Deployment and configuration of component-based distributed applications specification," 2006. [Online]. Available: http://www.omg.org/spec/DEPL/4.0/PDF

[9] A. Dearle, G. N. Kirby, and A. J. McCarthy, "A framework for constraint-based deployment and autonomic management of distributed applications," *International Conference on Autonomic Computing ICAC*, 2004.

[10] D. Sykes, J. Magee, and J. Kramer, "Flashmob: distributed adaptive self-assembly," in *Intl. Symp. on Software Engineering for Adaptive and Self-Managing Systems SEAMS*, 2011.

[11] F. Dressler, *Self-Organization in Sensor and Actor Networks*, Wiley, Ed., 2007.

[12] F. Nafz, H. Seebach, J.-P. Steghfer, G. Anders, and W. Reif, "Constraining self-organisation through corridors of correct behaviour: The restore invariant approach," in *Organic Computing A Paradigm Shift for Complex Systems*, 2011.

[13] L. Baresi, S. Guinea, and G. Tamburrelli, "Towards decentralized self-adaptive component-based systems," in *International workshop on Software engineering for adaptive and self-managing systems SEAMS*, 2008.

[14] R. Doursat, "Morphogenetic engineering weds bio self-organization to human-designed systems," *PerAda Magazine: Towards Pervasive Adaptation*, 2011.

[15] M. Ulieru and R. Doursat, "Emergent engineering: a radical paradigm shift," *Int. J. Auton. Adapt. Commun. Syst.*, vol. 4, no. 1, pp. 39–60, Dec. 2011.

[16] R. Nagpal, "Programmable self-assembly using biologically-inspired multiagent control," in *International joint conference on Autonomous agents and multiagent systems*, 2002.

[17] K. N. Lodding, "The hitchhiker's guide to biomorphic software," *Queue*, vol. 2, no. 4, pp. 66–75, Jun. 2004.

[18] J. Beal, "Functional blueprints: an approach to modularity in grown systems," in *Intl. Conf. on Swarm Intelligence*, 2010.