A Decentralized, Architecture-Based Framework for Self-Growing Applications

Ada Diaconescu
LIG laboratory, University of Grenoble
F-38041, Grenoble cedex 9, France
+33 4 76 63 55 66
ada.diaconescu@imag.fr

Philippe Lalanda
LIG laboratory, University of Grenoble
F-38041, Grenoble cedex 9, France
+33 4 76 63 55 60
philippe.lalanda@imag.fr

ABSTRACT
In large-scale, distributed software systems, an important management undertaking concerns the creation and runtime modification of application instances. This short paper proposes an autonomic framework that can produce and maintain coherent and adaptable application instances, in volatile execution environments. In the developed approach, decentralised, context-aware instantiation logic interprets a static architectural model and creates conforming instantiation solutions, customised for the current requirements. A local framework prototype implementing this approach was implemented based on a Service Oriented Component technology. Experimental results based on a Home Monitoring data-mediation scenario show the viability of the proposed solution and of the decentralised framework.

Categories and Subject Descriptors
D.2.11 [**Software Engineering**]: Software Architectures – *Domain-specific architectures, Patterns (e.g., client/server, pipeline, blackboard)*. D.2.13 [**Software Engineering**]: Reusable Software – *Domain engineering, Reuse models*.

General Terms
Design, Management, Reliability

Keywords
Autonomic instance management, decentralised instantiation logic, context-aware model interpretation, self-growing software.

1. INTRODUCTION
In large-scale, distributed software systems, an important management undertaking concerns the creation and runtime modification of application instances. In addition to the required deployment, installation and configuration operations, extensive administrative tasks involve determining the necessary types, cardinalities, interconnections and locations of created instances. During runtime, such administrative tasks must often be repeated in order to maintain the coherence of application instances in changing execution environments. For complex software systems, determining correct instantiation solutions in a dependable, timely manner can raise important management challenges.

Autonomic Computing can address this difficulty by introducing specialised utilities for determining application instantiation solutions and for performing the necessary operations for implementing these solutions. Scalability, resilience, dependability and adaptability are essential requirements for such utilities. Existing approaches generally use an application architectural model, interpreted by an automatic generator for deploying and instantiating distributed applications (e.g. [1] or [2]). Nonetheless, the available utilities are mostly centralised, meaning that the overall model of the targeted application must be available before the application can be instantiated and/or modified. Such approaches ensure the accuracy of the application management process but raise major difficulties when having to scale with increasing system sizes and modification frequencies.

This short paper proposes a decentralised solution for the autonomic creation and maintenance of coherent application instances, running in fluctuating environments. In the presented approach, context-aware instantiation logic automatically interprets a static, abstract architectural model and produces conforming application instances, customised for the current execution conditions. This solution generates adapted application instances, while the abstract architectural model and the interpretation logic remain unchanged. The instantiation control logic is decentralised, allowing the proposed solution to scale with application sizes, to adapt to evolving execution environments and to recover from the partial destruction of most of its parts. A framework prototype following this solution was developed and validated against a sample data-mediation application.

2. SOLUTION OVERVIEW
The proposed autonomic instantiation solution is based on two main elements – a static *architectural model* and a context-aware model *interpreter* [Figure 1]. The static *architectural model* formally defines the application's architectural constraints and possible architectural variations. All application instances must comply with the static application model. Namely, architectural models define the types, interconnections and general constraints that are common to all application instances. They do *not* describe the runtime architecture of an executing application instance. This means that an architectural model specifies neither the exact runtime instances, nor their precise runtime interconnections and platform assignations. A specific architectural model language is employed for formalising model specifications.

The context-aware *interpreter* automatically analyses the model and produces a compliant application instance, customised for the current execution environment. The interpreter's instantiation logic is decentralised. Namely, the interpreter consists of multiple *Instance Managers*, each one responsible for one application *type*. Instance Managers have identical implementations, merely their

ICAC'09, June 15–19, 2009, Barcelona, Spain.
ACM 978-1-60558-564-2/09/06.

associated types differ. Each Instance Manager *resolves* a fraction of the overall architectural model, by generating a partial application instance corresponding to its type. The entire application emerges from local collaborations between Instance Managers that solve adjacent architectural fractions. That is, each Instance Manager associated to a type *finds* or *creates* Instance Managers that are associated to the connected types, according to the architectural model. Once identified, neighbouring Instance Managers bind to each other and generate matching connections between their partial instance solutions. Based on this approach, the overall application instance and its underlying management support grow progressively from one or more *primary* Instance Managers. At runtime, the same logic is used to detect changes in application instances and to resolve potential inconsistencies.

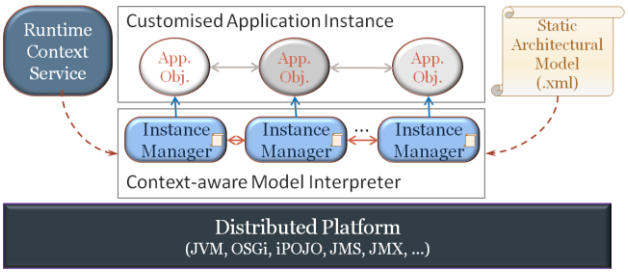

Figure 1: Solution Overview

Figure 1 shows the main architectural layers necessary for putting into practice the presented solution. Namely, the proposed framework executes on a specific technological *Platform*, which must be in place on all stations on which the targeted application can be instantiated. The actual framework comprises *Instance Managers* and *Runtime Context* entities. Instance Managers create and administer *Application Objects* (i.e. application instance parts) by following the constraints of a shared architectural model. The Runtime Context, available on each Platform, supplies Instance Managers with an up-to-date view of the local execution context (e.g. existing instances and available platform resources).

3. DECENTRALIZED MECANISMS

The framework's overall behaviour is based on a number of simple, decentralized mechanisms, which ensure the correctness and the coherence of the emerging application instances. These processes are similar to the ones presented in biologically-inspired engineering literature (e.g. [3] or [4]).

Event propagation: enables event diffusion over a certain scope, where scopes can be defined in various ways (e.g. geographical or network-specific location; distance from a source). This mechanism provides the communication support on which most of the other decentralised mechanisms are based.

Instance density detection: determines the number of Application Objects of a certain type within a given scope. Instance Managers signal the presence of associated Application Objects via specific *markers*. Event propagation diffuses each marker throughout the marker's scope. Runtime Contexts aggregate intercepted markers and calculate local instance densities. Instance Managers adjust instance densities to current conditions, while ensuring they conform to model constraints.

Competition for action: limits the number of Instance Managers that can perform a certain action within a certain scope (e.g. create or destroy instances of a type). Specifically, all Instance Managers within the scope compete against each other by means of a random countdown. The winning Instance Manager performs the action and inhibits the other Instance Managers.

Self-replication and self-destruction: enable the creation and the destruction of Instance Managers of an existing type. The goal is to regulate the number of instances of each type, depending on the current runtime context and on the imposed model constraints.

Activity desynchronization: prevents Instance Managers from simultaneously resolving their local models and hence potentially causing resource consumption peaks or overflows. This process selects various waiting periods (random or context-based), which precede the Instance Managers' reactions to certain events.

Periodic model conformance verification: triggers the instance creation and verification process at given intervals. This process is important when the events signalling dynamic changes in the application instance are lost (e.g. in an overloaded environment).

4. CURRENT AND FUTURE WORK

A framework prototype was implemented based on a Service Oriented Component technology – iPOJO[1], which is an OSGi[2] extension developed in the Adele team and made available as an Apache open-source project. The prototype was initially limited to a single Platform, an associated local Runtime and a core set of the decentralised mechanisms: local event propagation, density detection, competition for action and activity desynchronization. The current prototype was locally tested on a Home Monitoring data-mediation application. Starting from a set of seed instances (i.e. sensors for home resource consumption), the framework initially generated a data-mediation hierarchy, conforming to a given abstract architecture. Next, the framework extended the existing hierarchy so as to integrate dynamically added sensors. Finally, the framework recreated the hierarchy when the majority of its nodes were accidentally destroyed. These initial results are promising, as they indicate the viability of the adopted architecture-based, decentralised approach.

Immediate extensions will focus on: implementing and testing further decentralised mechanisms; adding support for distributed Platforms; performing the actual Application Object instantiation and binding operations (simulated in the current version); and extending the current model specification language.

5. REFERENCES

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

[2] Sicard, S., Boyer, F. and Palma., N. D. 2008. Using components for architecture-based management: the self-repair case. Proceedings of the 30th International Conference on Software Engineering (ICSE), pp 101–110, 2008.

[3] Nagpal, R. 2003. A Catalog of Biologically-inspired Primitives for Engineering Self-Organisation. Workshop on Engineering Self-organising Applications, Autonomous Agents and Multiagents Systems Conference (AAMAS)

[4] Tempesti, G., Mange, D. and Stauffer, A. 1997. A Self-Repairing Multiplexer-Based FPGA Inspired by Biological Processes. Journal of Systems Architecture: the EUROMICRO Journal, v.43 n.10, p.719-733, 1997

[1] iPOJO Project: www.ipojo.org

[2] OSGi Alliance: www.osgi.org