# Architectural Integration Patterns for Autonomic Management Systems

Sylvain Frey [1] [2], Ada Diaconescu [1] , Isabelle Demeure [1]

[1] Infres department
CNRS-LTCI, Télécom-ParisTech
Paris, France
{first name}.{last name}@telecom-paristech.fr

[2] ICAME department
EDF Research & Development
Clamart, France
{first name}.{last name}@edf.fr

***Abstract*—Large-scale, dynamic, distributed and open autonomic systems pursuing multiple, possibly conflicting goals are difficult to design, implement and maintain. Dealing with the complex issues that such systems raise requires complex, adaptive management logic. This paper focuses on the *integration* of autonomic management resources as a key feature for building complex autonomic systems. This paper's first contribution is an investigation, via a simple model, of integration issues in autonomic management systems. The discussion is illustrated via a reference use case involving smart homes connected to a micro smart grid. The second and main contribution consists in a collection of architectural design patterns, described following the classical form used in software engineering. The proposed patterns address different classes of integration problems, mostly concerned with conflict resolution. They are comparatively evaluated via a set of quality attributes and exemplified via reference conflict situations in the use case. Several possible extensions are subsequently identified, including various pattern compositions. The presented research is part of a more general, broader approach towards a generic, reusable framework for designing, developing and maintaining autonomic systems.**

***Keywords- autonomic computing; self-* systems; complexity; design patterns; integration; framework; software engineering.***

## I. INTRODUCTION

Complex systems require complex autonomic management applications to administer them. Large scales, dynamism, distribution, heterogeneity, openness and multiplicity of (possibly conflicting) goals are common factors of complexity in modern computer systems. Designing, developing and maintaining autonomic management applications for such systems is a difficult and costly task, at best. The availability of generic, reusable architectures, frameworks and methodologies would provide significant help for addressing this problem. While partial solutions do exist (e.g. [4,5,6,7,8]), considerable progress is still required towards providing a comprehensive, seamlessly reusable support for autonomic software.

The approach presented here is to consider system *integration* as an essential requirement and challenge facing the autonomic computing community [9,12]. Firstly, the paper describes a general approach for designing complex management systems based on a modular, flexible and generic model. This approach is illustrated with a concrete use case from the micro smart grid domain. Integration issues – mostly "conflicts" – are highlighted as a major obstacle to designing proper autonomic systems. Secondly, the main contribution of this paper comes as a collection of *architectural design patterns* for system integration, in the context of the presented autonomic system model. Following the classical form used in software engineering, these patterns identify an application *context*, describe an integration *problem* and expose a *solution* that is exemplified in the proposed use case. Patterns are compared based on a set of quality attributes and several pattern combinations are proposed.

The structure of this paper is the following: after a review of related work in part II, part III exposes a generic model for autonomic systems and discusses integration issues. Part IV presents a reference use case illustrating the autonomic model in concrete situations. Part V describes the proposed integration patterns for management systems as traditional software design patterns, with illustrations in the use case and evaluation with respect to pre-defined quality attributes. The final section concludes the paper and discusses further research perspectives.

## II. RELATED WORK

Organisational patterns in complex systems have been investigated in various research domains, most notably including multi-agent systems [10], robotics [27], service-oriented enterprise systems [13] or bio-inspired applications [19]. Seminal works in artificial intelligence such as [25,27] established the need for sophisticated architectures for designing and building complex systems.

The multi-agent domain shares certain paradigms with the autonomic computing domain and hence represents an important source of inspiration for integration solutions. For instance, [10] provides a rich series of agent organisations, with an analysis of their respective features, strengths and weaknesses. While certainly useful, these organisations are strongly coupled with the agent model which is not straightforward to apply in the context of most industrial autonomic applications [35]. A fortiori, autonomic management systems feature specific organisational issues (such as the management conflicts described below) that the agent model is not specialised in. Similarly, [18] defines and analyses "interaction patterns" in multi-agent populations. These patterns are based on specific features of the agents considered, such as their tendency to compete or collaborate with their peers. Therefore their applicability is limited to very special cases of autonomic system designs (e.g. decentralised collaboration amongst autonomic managers).

Adjacent to the multi-agent and autonomic domains, contributions such as [19,20,31] formulate biology-inspired patterns based on low-level message-based communications (data diffusion, replication, repartition) focusing on concerns such as robustness, dynamism and scalability.

In industry, a significant number of solutions have been proposed in answer to integration issues. Multiple service

and component-oriented technologies such as CORBA [36], Java EE [37], .NET [38], Android [39], OSGi [13] or iPOJO [14] targeted various properties such as modularity, code reuse, loose-coupling or flexible deployment in large-scale, open computer systems. Service-oriented architectures [13,21,22] address interoperability and/or dynamicity issues. While such approaches provide good integration solutions at a basic, technological level, they remain too generic for addressing domain-specific integration problems, such as those occurring in autonomic systems (e.g. goal-level conflict resolution).

Ceylon project [4] is a previous work relying on such technological base for providing a domain-specific framework for complex autonomic management systems. Ceylon proposed a generic solution for dynamically integrating autonomic management resources into complete management loops. However, Ceylon covered solely some of the integration patterns presented here. This paper extends and generalises Ceylon's approach with a wider investigation of integration solutions in the form of architectural design patterns. While many of such solutions have already been proposed and/or instantiated in various domains (e.g. multi-agent systems, robotics, service-oriented enterprise systems) their applicability to the autonomic computing domain remains to be identified and documented.

## III. GENERIC MODEL AND INTEGRATION CONFLICTS

### A. A generic model for autonomic systems

The model presented here is solely introduced as a basis for illustrating integration notions, issues (part IV) and solutions (part V) in autonomic management systems.

The model makes a clear separation between managed application resources and Autonomic Management Resources (AMR), that belong to the application layer and to the management layer, respectively. AMRs are specialised components [4] compliant with the following "classic" model [16]: the AMR *core* implements various autonomic management functionalities, such as special-purpose Monitoring, Analysis, Planning and Execution defined in the MAPE-K architecture [1,2]; the AMR *container* embeds non-functional features such as communication handlers or integration resources (cf. section C below). Communications are based on models that favour loose coupling between AMRs (e.g. publish/subscribe messaging or dynamic bindings between standardised interfaces). For the sake of simplification and without loss of generality, this paper considers that all communications are based on generic "messages".

In addition to explicit communication, the model also considers indirect influences between resources. For instance, an application resource (e.g. a heater) may influence another application resource (e.g. a thermometer) exclusively via the environment (e.g. via the temperature of a room). Fig. 1 introduces a graphical representation formalising the model.

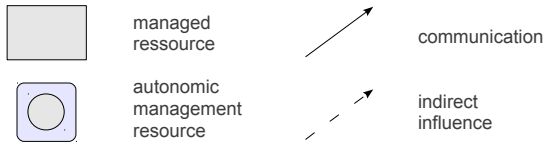


*Figure 1: basic elements of autonomic systems.*

Based on this model, Fig. 2 depicts a "classic" autonomic element, with a single AMR performing the entire autonomic management loop for a single managed resource.

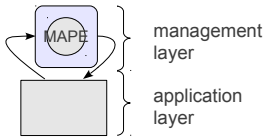


*Figure 2: representation of an autonomic element.*

### B. Complex autonomic management layers

Complex, adaptive management layers are required for administering complex, ever-changing application layers – i.e. involving a high number of heterogeneous, dynamic, unpredictable application resources and multiple conflicting management objectives. Building such complex management layers can be accomplished by (dynamically) integrating a reusable set of simpler management components, or resources, as proposed for example in the Ceylon project. Ceylon [4] considers open, flexible sets of loosely-coupled AMRs implemented with advanced service-oriented component frameworks [13,14,15]. Ceylon's AMRs are highly modular, reusable components, each one implementing one or several of the MAPE functionalities. Using dynamic deployment, binding and reconfiguration, Ceylon allows combining AMRs opportunistically for forming complete management chains.

For instance, multiple Analyser instances with different performances and execution costs may be run in parallel, started or stopped, allowing the selection of the most appropriate one according to available resources, time limits or Quality of Service criteria. Based on these principles, the management layer can be dynamically extended and adapted to a wide range of managed systems, while favouring AMR reuse and separation of concerns. In particular, AMRs can participate simultaneously in several management loops within the same management layer. A complex management chain à la Ceylon is represented in Fig. 3.

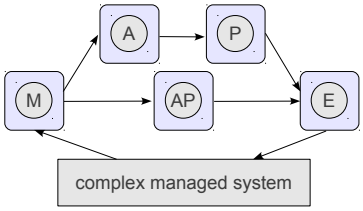


*Figure 3: complex management chain for a complex managed system [4].*

Integration becomes a key concern when adopting this type of approach for building complex management layers. Specifically, integration may occur "internally" – i.e. integrating AMRs in order to obtain complete MAPE loops within one autonomic system (Fig. 3); and "externally" – i.e. integrating independent autonomic management systems to form a coherent, global autonomic system (Fig. 4).

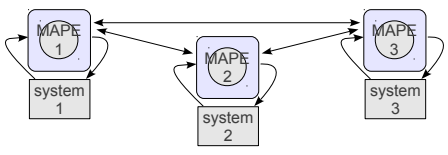


*Figure 4: integration of heterogeneous autonomic systems.*

The integration situations described in Fig. 3 (intra-system integration) and Fig. 4 (inter-systems integration) are slightly different. For the rest of this paper, it will be considered that "AMR integration" encompasses both these situations, whether or not the management layer comprises a unique system or several ones.

### C. *Integration issues in complex management layers*

While complex management layers featuring large sets of heterogeneous, dynamic AMRs are a necessity, they also raise a major and difficult issue: how to actually integrate AMRs in order to obtain consistent, coherent management systems, capable of reaching their administrative goals. Integration comprises several inter-related sub-problems, including communication, compatibility, synchronisation and conflict resolution. This paper focuses on conflict resolution, as one of the most difficult issues specific to the autonomic domain. Further integration concerns will be addressed in future extensions.

In the context of this work, “conflicts” are defined as the clashing of contradictory management control flows, as shown on Fig. 5 and 6 (conflicting AMRs are marked with an “X”). Several situations may lead to such conflicts. Most commonly, several AMRs may try to act on the same managed resource, providing conflicting control commands and leading to incoherent behaviours (left case in Fig. 5). Another conflict situation that is typical to the presented model may occur within integrated management chains, when several communication flows converge onto a single AMR (right case in Fig. 5). For instance, two Planning AMRs are deployed on the same application resource and send contradictory commands to the same resource Executor. Or, two Analysers send contradictory reports to the same Planner in the management chain.

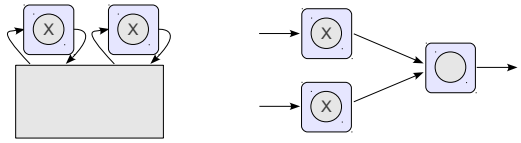


*Figure 5: direct conflict cases, resource level (left) and management level (right).*

In addition to such direct conflict situations, indirect conflicts can be derived as caused by transitive influences at the application level (left on Fig. 6) or in the management chain (right on Fig. 6). For instance, two devices influence temperature in a room, therefore their temperature managers are potentially in conflict. Or, a conflict between two Planners in the management chain may result from an upstream conflict between two Analysers.

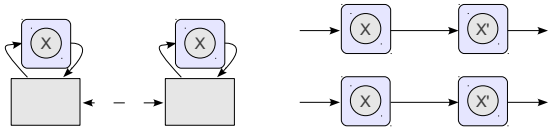


*Figure 6: indirect conflict cases, resource level (left) and management level (right).*

Integrating autonomic systems requires resolving their integration conflicts. Still following the principle of separation of concerns, special-purpose integration resources and specific integration communications are added to the model, as an addition to core management functionalities (i.e. MAPE functions). Integration resources perform specific conflict-resolution functionalities, which will be described in pattern definitions. Fig. 7 shows a formal graphical representation of integration resources.


*Figure 7: integration-specific elements of the proposed model.*

There are several ways integration resources can be inserted into the management layer. As shown on Fig. 8, they can be: deployed as stand-alone components, or AMRs (left in the figure); or injected into an AMR container (centre); or mixed with management logic (right). Injecting integration logic into an AMR container enables it to intercept and alter incoming and outgoing communications, while remaining completely transparent to the core management logic.

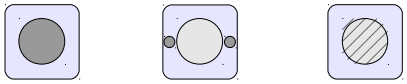

*Figure 8: possible deployment of conflict resolution resources.*

Next section exposes a reference use case to illustrate an utilisation of our formalism for modelling concrete autonomic management systems and analysing conflict situations. Possible solution semantics to these conflicts are then proposed.

## IV. Analysis of conflicts in a reference use-case

This section presents a sample use case formulated with the proposed model. Conflicts occurring in this use case are reused in part V as examples for introducing the proposed conflict-resolution patterns.

### A. *Autonomic management in smart houses and grids*

Let us consider a smart house connected to a smart electrical grid. Sensors allow measuring environment parameters, such as room temperatures and grid load. For the sake of simplification, we consider that the grid could be either under high, normal or low load, where the load represents the ratio between consumption and production in the grid. High load signifies that production should be increased and/or consumption decreased; low load implies the contrary. A grid undergoing abnormal loads may lead to inconvenient energy bills and possibly to blackouts. In this scenario, the electrical equipments of the house are expected to participate in load regulation by lowering their consumption when the grid is overloaded.

In one of the smart house's rooms, an electrical heater has its emission power controlled by two AMRs (cf. Fig. 9). A temperature AMR (“AP heaterTemp”), sensitive to the temperature in the room and to user instructions, tries to maintain optimal comfort. At the same time, an energy AMR (“AP heaterEnergy”), sensitive to the grid load, tries to reduce the heater's consumption whenever load peaks occur. In the house's kitchen, a smart refrigerator comes equipped with an energy AMR (“APE refrigEnergy”). In case of high

load, this AMR can shut down the refrigerator, stopping its consumption for a limited amount of time (typically one hour, once a day) without threatening its food content.

Fig. 9 represents the use case using the proposed model, with several interconnected MAPE-like AMRs. Several indirect influences are also exemplified. E.g., the heater indirectly influences the room thermometer via the room's temperature. Similarly, the refrigerator and the heater indirectly influence the load meter through their consumptions on the electrical grid.

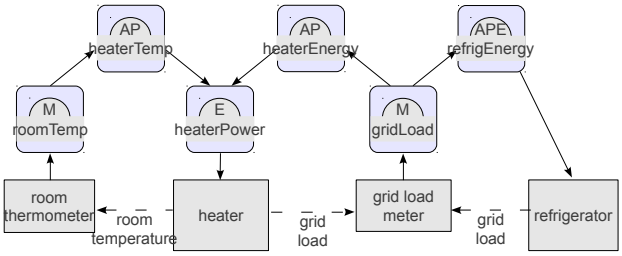


*Figure 9: autonomic management in the smart home and grid use case.*

### *B. Semantics of the autonomic systems*

Considering the described devices and corresponding AMRs, two conflicts can be identified and require preventive resolution.

#### *1) Heater management conflict*

The two Analyser-Planner AMRs on the heater ("AP heaterTemp" and "AP heaterEnergy") are conflicting as they attempt to control the same "E heaterPower" Executor for setting the heater's emission power. For instance, on a cold winter night, the "AP heaterTemp" may want to set a high emission value in order to maintain comfortable conditions. On the other hand "AP heaterEnergy" may detect high consumption levels on the grid – since many electrical heaters would work hard in the neighbourhood – and try to lower the heater's power.

This conflict happens because the two "AP" AMRs follow fundamentally incompatible goals: temperature management and energy management. Therefore, an explicit solution to this conflict must be provided for the entire system – heater and AMRs – to behave "properly". Certainly, such solution will highly depend on the expected system behaviour from the user perspective. We propose here two business-level approaches for the solution:

**"all or nothing"**: only one of the two "AP" AMRs controls the heater at any given moment; the other one's command is ignored;

**"compromise"**: an intermediate value is used for setting the heater's emission power; this value is the mean of the two "AP" AMR advices.

The "all or nothing" solution solves the conflict by neglecting one of the objectives against the other, whereas the "compromise" one tries to fulfil both of them at the same time – with the risk of satisfying neither.

It is important to ensure that the behaviour of the management resources is compliant with the semantics of the conflict resolution. Since the latter decouples management decisions (i.e., heater reconfiguration plans) from effects at the resource level (i.e. the actual value of the heater's emission power), the AMRs must be able to support such situation. Indeed, if one AMR were adaptive, it could compensate for and so cancel the effects of conflict resolution. If both AMRs were adaptive, conflict resolution may work but prove inefficient as both AMRs constantly try to compensate in opposite directions. Hence, the compatibility between resolution strategies and targeted AMRs to integrate must always be considered.

For instance, let us suppose that at time T the "AP heaterTemp" AMR proposes a value of 20 for the heater's power (on an arbitrary scale), the "AP heaterEnergy" AMR proposes 18 and a "compromise" resolution mechanism eventually sets a 19 value on the heater. An adaptive "AP heaterTemp" AMR, monitoring that its temperature objective (20) is not reached, might decide at T+1 to compensate and propose a value of 22 instead, yielding a final power at 20 and cancelling the energy AMR's influence, which is in this example passive for the sake of demonstration. In the case where both AMRs are adaptive, divergent behaviours can be expected from the AMRs, as they each pull the value in its own direction.

For this use case, let us consider that the management resources, in particular all the analyser-planner (AP) AMRs, are purely reactive and non-adaptive. Therefore, they will be compatible with "all or nothing" semantics and will not cheat "compromise" solutions. Other scenarios will be considered to include adaptive AMRs in future work.

#### *2) Electricity management conflict*

The two energy AMRs operating on the heater ("AP heaterEnergy") and on the refrigerator ("APE refrigEnergy") are in conflict, since both devices interact with the electricity grid and influence its load with their consumptions. However, this kind of conflict is different from the heater management one, since here the two AMRs follow the same objective – energy regulation, while controlling different managed resources.

During a load peak, one of the AMRs applying consumption reductions may be sufficient to bring back the load to normal levels, as it indirectly relieves the other AMR from doing so. On the contrary, if one AMR allows high consumption the other one may be forced to apply consumption reductions that could have been avoided otherwise. Worse still, the two devices reducing their consumption at the same time might equally bring undesirable consequences, such as after-effect load surges.

Therefore, a conflict resolution mechanism is necessary for the two energy AMRs to integrate properly with each other and maintain a consistent load. Without going too deep into details, two features of this resolution are specifically required:

**"helpfulness"**: whenever a device could help load control by reducing its consumption without harming its functioning, this device's energy AMR should command so;

**"coordination"**: in case both devices are available for reducing their consumption, they should not trigger it at the same time. Instead, they should coordinate and fairly designate a first energy saver that will actually reduce its consumption, and a second one that will do so only in the case where load does not go back to a normal level.

This description of load management in an electricity grid is arguably oversimplified, yet it shows some actual issues, fairness and synchronisation, that the smart grid community is facing [23]. [28] presents previous work on more comprehensive smart grid scenarios and load management, developed in collaboration with EDF (French national electricity company). Part V presents possible solutions for achieving this load management approach.

## V. DESIGN PATTERNS FOR CONFLICT RESOLUTION

This section presents a number of generic design patterns for addressing conflict resolution in the autonomic management layer. Each pattern presentation starts with elements of *context*, showing the particularities of the *conflict situations* the pattern addresses. Then the pattern *solution* is described using the notations introduced in part III, with conflicting AMRs identified by an "X". The impact of conflict resolution on the management system is evaluated with respect to a set of quality attributes, including:

**overhead**: the additional computations and communications that the pattern solution introduces in the management chain.

**safety**: the assurance that neither managed resources nor AMRs evolve to an undesirable state.

**conceivability**: the difficulty in designing and developing the pattern solution, particularly, with respect to integrating legacy AMRs and handling different management goals.

**robustness**: the ability of the pattern solution to resist to and recover from the faults of its conflict resolution resources, as well as to handle incorrect input.

**evolvability**: the ability of the pattern solution to adapt to changes of the autonomic system, in particular to the removal and arrival of conflicting AMRs.

**scalability**: the ability of the pattern solution to scale up to a large number of conflicting AMRs.

For each pattern an example application is proposed in the context of the presented use case (part IV). This section ends with a discussion on patterns comparisons and possible combinations.

### A. Monolith pattern

#### 1) Context

Only a few simple management resources are conflicting, the conflicts and their solution are clearly identified and unlikely to change in the future. Decoupling management logic and conflict resolution logic is not necessary. On the contrary, a separation would induce unnecessary complexity and overheads (i.e. over engineering).

#### 2) Description

*Figure 10: Monolith pattern.*

The logic of AMRs and of conflict resolution resources is merged into a monolithic management solution.

#### 3) Evaluation

Being a strongly-coupled solution, the Monolith can be globally optimised for the specific logic of conflicting AMRs and conflict resolution strategies. Hence, conflict resolution overheads can be minimised. Moreover, the solution is predictable and may be safety-proven.

However, the Monolith can only be applied to moderately complicated cases that are unlikely to evolve (few simple conflicting AMRs). Otherwise this solution would become hard to conceive and maintain, since introducing new management logic or modifying the conflict resolution strategy would imply recoding the Monolith. Finally, as a centralised entity, the Monolith represents a bottleneck for communications with the managed system and other autonomic systems.

#### 4) Example application within the smart home use case

The Monolith pattern could indeed be applied to solve the heater management conflict: the two "AP" AMRs ("AP heaterTemp" and "AP heaterEnergy") are merged with an explicit resolution mechanism, implementing either the "all-or-nothing" or the "compromise" solution.

On the other hand, a Monolith for load regulation in the house might prove to be an undesirable solution. However predictable, such a Monolith would be difficult to conceive with respect to the specificities of each device and their management resources. Furthermore, the Monolith being neither adaptable nor scalable would make it difficult to add new "smart" devices to the energy management system.

### B. Dealer pattern

#### 1) Context

The execution of conflicting AMRs is triggered by management messages – i.e. AMRs are reactive, event-driven components. The autonomic management architecture is such that the incoming communication flows triggering these conflicting AMRs pass through a single upstream AMR (e.g. a set of conflicting Planners fed by a single Analyser).

#### 2) Description

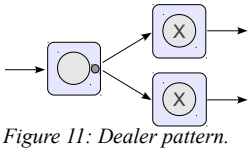


*Figure 11: Dealer pattern.*

Since the flow of messages pass through a shared resource before triggering conflicting AMRs, there is an opportunity at this unique point to filter outgoing messages, so as to avoid triggering conflicting AMRs down the management chain. As a result, the filtering component (the "Dealer") prevents conflicts by starving all but one of the conflicting AMRs. This implies that a single path is being selected for execution amongst all conflicting possibilities in the management chain. At the design level, the conflict-resolution logic of the Dealer can be introduced as a special-purpose handler in its container.

#### 3) Evaluation

The Dealer has a low overhead, since it prevents conflicts altogether and hence avoids useless computations and communications. It does not require the introduction of new components since the resolution code is introduced in the container of an existing AMR. As the resolution logic capitalises on the AMRs' reactivity to input messages, no modifications are required on the conflicting AMRs. A possible drawback, since the Dealer acts upstream the

conflicting AMRs, it may be unable to detect faults or check the actual validity of the AMRs it dispatches messages to. The Meta-Manager (section G below) may be combined with the Dealer to avoid such situations.

The Dealer can be seen as a single point of failure of the management chain and a bottleneck for communications, with consequences on robustness and scalability. However, since the Dealer relies on a central upstream AMR that already exists in the management chain the aforementioned limitations are the consequences of the management chain design and not of the pattern solution.

*4) Example application within the smart home use case*

A Dealer can be deployed in the grid monitoring AMR ("M gridLoad" on Fig. 9) that produces load information for energy AMRs ("AP heaterEnergy" and "APE refrigEnergy"). Hence, this Dealer can decide to route "high load" messages only to one of the conflicting "energy" AMRs, preventing the other one from triggering undesirable consumption reductions.

One could also imagine an extended Dealer-based solution to the heater management conflict. Namely, a Dealer can be added as an additional component to the heater's management chain, intercepting all incoming messages addressed to the conflicting "AP" AMRs ("AP heaterEnergy" and "AP heaterTemp"), starving one of the two and feeding the other one normally. This architecture would only allow a "all-or-nothing" solution to the conflict, very close to the Controller pattern shown later in this section. This would be an extended Dealer version, since a special-purpose component would have to be added to the management chain.

Within the Ceylon project, an early prototype version [3] provided a generic mechanism for conflict resolution that was similar to the Dealer, in that all AMR messages had to pass via common Event Bus. Conflict resolution was executed at this level by filtering out conflicting messages.

## C. Aggregator pattern

*1) Context*

The autonomic architecture is such that all output management flows of conflicting AMRs pass through a single AMR (e.g. several conflicting Planners connected to a single Executor).

*2) Description*

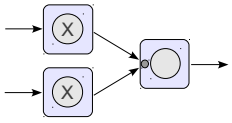

*Figure 12: Aggregator pattern.*

Contradictory orders or conflict-triggering messages are intercepted by the conflict resolution resource – the "Aggregator" – which in turn produces a coherent, conflict-free solution as its output. The Aggregator's conflict resolution logic can rely on an abstract message-filtering process (e.g. select higher priority messages or compute a weighted sum), or be inspired by additional business expertise. Similarly to the Dealer, the Aggregator logic can be encapsulated as a handler in the shared AMR's container.

*3) Evaluation*

The Aggregator integrates seamlessly into the management chain since it does not require modifying any of the AMRs's business logic. Since it is placed downstream from conflicting AMRs, it can ensure that the final result is coherent, in particular in case an AMR is faulty or produces unexpected messages.

The Aggregator's synthesis of conflicting communications introduces variable overheads, depending on the actual conflict resolution logic used. The Aggregator does not prevent conflicting AMRs to execute, which may allow for wasteful computations in case the synthesis uses the "all or nothing" strategy (cf. part IV). Being centralised (with respect to one conflict-resolution situation), it can be viewed as a single point of failure and a communication bottleneck. However, as for the Dealer pattern, this is a feature of the management chain design and is independent from conflict resolution.

*4) Example application within the smart home use case*

An Aggregator is a straightforward solution to the heater management conflict: the heater's power emission Executor ("E heaterPower" on Fig. 9) is extended with an additional conflict resolution logic implementing either the "all-or-nothing" or the "compromise" solution. Such a smart heater using an Aggregator pattern has been previously investigated in [29].

In the energy management case, the conflicting flows (i.e. the execution orders sent to the heater and the refrigerator) do not converge to a single point and hence the Aggregator pattern is not applicable.

## D. Controller pattern

*1) Context*

Conflicts involve extensive numbers of AMRs, which share neither any input nor any output flows (i.e. no single upstream or downstream AMR). Neither the Dealer nor the Aggregator pattern can be applied in this case. Yet, conflicting AMRs are still reactive to messages and a conflict resolution strategy based on message filtering can be appropriate.

*2) Description*

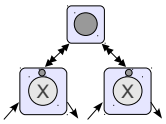

*Figure 13: Controller pattern.*

A special-purpose centralised component – the "Controller" – prevents clashes among conflicting AMRs by diverting and controlling their incoming and/or outgoing management flows. Several Controller variants are possible, depending on the way in which management flows are diverted. In the simplest variant, the Controller is implemented within the middleware infrastructure ensuring inter-AMR communication (e.g. as a special-purpose service within an Enterprise Service Bus). It can consequently intercept and control – e.g. filter-out – any management flow, thus preventing the simulation activation of conflicting AMRs [3]. In a more flexible Controller variant, conflicting AMRs are allowed to receive all messages but must ask permission from the Controller before executing their management logic [4]. In this variant, only management flows that are known to trigger conflicting AMRs are actually diverted and controlled, by configuring the AMRs that send or receive them to communicate with the

Controller. This facility limits conflict-resolution overheads to conflicting AMRs. Similarly, conflicting AMRs may be allowed to execute, but be obliged to ask for the Controller's permission before sending any output management messages. In either variant, the Controller does not take part in the actual autonomic management business process, since it does not act directly on managed application resources. Its purpose is exclusively related to conflict resolution. For instance, projects such as [5,8] feature Controller-like dedicated conflict resolution resources.

*3) Evaluation*

The Controller can explicitly ensure desirable properties of the AMRs it controls, such as forbidding their simultaneous execution or ensuring a priority ranking of their management messages. These properties can be safety-proven.

The Controller introduces a new special-purpose component and requires additional communication for conflict resolution. While this introduces certain overheads, the incurred delays can be limited to AMR groups that were identified as potentially conflicting.

The interdependence level between the Controller and the controlled AMRs is variable, depending on how specific the Controller logic is with respect to the AMRs business logic. Therefore, the system's designer can adapt their solution to trade-offs such as precision of conflict resolution vs. evolvability.

In the first Controller variant, the Controller's decision logic may become difficult to figure-out for large numbers of conflicting AMRs. In the last two Controller variants, conflicting AMRs are forced to adopt a specific behaviour (asking permission), which can be viewed as intrusive. At the same time, the actual AMR logic involved in Controller communication can be implemented as a special-purpose handler in the AMR container (Fig. 13). In all cases, the Controller's centralisation introduces a single point of failure and a communication bottleneck, potentially hindering scalability with the number of conflicting AMRs. The Meta-Manager pattern (section G below) can be introduced in this case to alleviate such situations.

*4) Example application within the smart home use case*

A Controller can be used in the heater conflict situation of the smart-home use case. More precisely, the two conflicting "AP" AMRs ("AP heaterTemp" and "AP heaterEnergy" on Fig. 9) are configured to ask permission to an additional Controller component, either before executing (i.e. just after receiving monitoring messages), or before sending their plans to the Executor. In this solution, the Controller only allows for an "all-or-nothing" resolution strategy. Finally, the Controller pattern may be an overkill for the heater's simple conflict and was only exemplified here for illustration purposes. In [4], Ceylon project provides a more comprehensive example of a Controller application.

## *E. Hierarch pattern*

*1) Context*

Conflict resolution is a feature of the management layer, and it is not possible, or desirable, to completely separate management logic from conflict resolution logic. Multiple conflicting AMRs provide fine-grained management functions (e.g. M, A, P, E or simple MAPE loops), while the resolution logic handling these AMRs requires a higher-level and more abstract overview of the management process.

*2) Description*

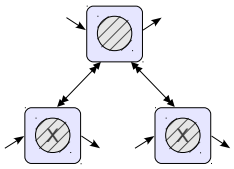

*Figure 14: Hierarch pattern.*

Conflict resolution is provided by a higher-level AMR – the "Hierarch" – featuring both management logic and conflict resolution logic. At the autonomic management business level, the Hierarch has a broader, more abstract view of the managed application and can thus take better-informed decisions at the global level. At the conflict-resolution level, the Hierarch can capitalise on its business-specific knowledge acquired from its global position and scope (in comparison to the conflicting AMRs) in order to take better-informed, globally-optimised resolution decisions. To resolve a conflict, the Hierarch uses special-purpose AMR interfaces to act directly on the conflicting AMRs, in terms of authoritative commands or optional recommendations.

The Hierarch differs from the Controller in that it is not a dedicated conflict resolution resource, but a higher-level manager that features conflict resolution amongst other capabilities. Rather than intercepting the management flow among conflicting AMRs, as in the Controller's case, the Hierarch executes in parallel with the AMRs and only intervenes when a conflict is detected or predicted. Finally, while in the Controller's case the AMRs' core business logic was completely agnostic to the Controller's intervention, in the Hierarch case the AMRs core business logic may have to take into account specific recommendations or orders from the Hierarch.

From a certain perspective, the Hierarch pattern can also be compared with the Monolith, since management business logic and conflict resolution logic are merged. However, while the Monolith merges the logic of conflicting AMRs with the resolution logic, the Hierarch maintains conflicting AMRs independent (with all the modularity and flexibility advantages provided by that approach). Conversely, the Hierarch capitalises on the availability of an already existing high-level AMR to place resolution decisions within the context of more knowledgeable management logic.

Many research proposals within the autonomic computing domain have proposed hierarchical management solutions similar to the Hierarch pattern, e.g. [33,34].

*3) Evaluation*

The Hierarch involves little overheads since it introduces no special-purpose integration component and since it does not divert "normal" AMR management flows. Since its resolution logic is mingled with and reliant upon its management logic, the Hierarch's evolvability depends on the evolvability of its management logic. With this consideration in mind, the Hierarch should withstand the addition and removal of various AMRs with minimal required modifications on the integration infrastructure. Since it is centralised, the Hierarch's scalability may be limited by the number of AMRs it can supervise. Nonetheless, since the Hierarch is based on an available high-level AMR, this limitation is already inherent to the

pre-existing autonomic system design. With respect to robustness, a failure of the Hierarch component would remove conflict-resolution support, while leaving the lower-level autonomic management process unaffected.

*4) Example application within the smart home use case*

A Hierarch can be used to solve the energy-regulation conflict. In this case, the Hierarch represents a central coordinator, relying on a global view of energy consumption/production for controlling energy savings. The Hierarch is deployed as a stand-alone AMR and sends energy-saving recommendations and/or commands to conflicting "energy" AMRs ("AP heaterEnergy" and "AP refrigEnergy" on Fig. 9). Capitalising on its global, higher-level view of the house's and the grid's load, it can predict future load profiles and propose or impose better-informed solutions to individual energy AMRs. Supposing a reasonable number of devices in a house, scalability issues inherent to centralisation should not be too critical, a priori. As a more significant example, the Hierarch can be beneficially applied for load management at larger grid scopes (i.e. neighbourhood, city or country), in order to provide high-level energy-management directives based on a global grid-load model. In this case, scalability issues would require a finely-designed hierarchy with progressively increasing management abstraction levels.

### *F. Collaboration pattern*

*1) Context*

The autonomic management logic must address multiple administrative concerns, while remaining highly-adaptable to frequent changes in managed resources, available AMRs and targeted goals. A large, highly-dynamic and open set of AMRs must be dynamically integrated into coherent management chains. Centralised or hierarchical solutions cannot meet scalability requirements and Monolithic conflict resolution cannot cope with frequent changes of conflicting AMRs.

*2) Description*

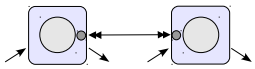

*Figure 15: the Collaboration pattern.*

Each conflicting AMR embeds its own conflict resolution logic, implementing a completely decentralised algorithm. Overall conflict resolution results from collaborations, negotiations and/or competitions among conflicting AMRs. Several Collaboration variants can be envisaged, depending on the nature of the AMR coordination protocols. For instance, AMRs may use inhibition or voting algorithms to designate a "conflict winner" that takes the conflicting management decision. Or, AMRs may negotiate a compromise solution. From a design perspective, conflict-resolution logic is encapsulated as a special-purpose handler within each AMR container. As before, this clearly separates core AMR business logic from the resolution logic controlling its execution.

A possible variation of this pattern is the *stigmergic collaboration* pattern: instead of interacting directly with explicit communications, the conflicting AMRs influence each other via an indirect influence in the environment.

*3) Evaluation*

Being distributed and totally decentralised, a Collaboration is likely to be highly robust – featuring no single point of failure – and scale up well to a large number of conflicting AMRs. Open Collaborations can allow great evolvability, supporting the conflicting AMRs' churn and their behaviour evolutions over time.

However, decentralised systems are also difficult to conceive and may show a high communication overhead. Complete control, overall optimisation and predictability of global system behaviour can also be extremely difficult to attain [11,24]. Therefore, a Collaboration may be inappropriate in case high reactivity is required from the management system. Global safety properties of the Collaboration might prove hard to figure out and to implement. The solution is rather intrusive since it imposes a specific behaviour (i.e. collaborating, negotiating or competing) on the conflicting AMRs.

*4) Example application within the smart home use case*

A Collaboration is an extremely robust and scalable solution to the energy management conflict. Decentralised, peer-to-peer algorithms such as "firefly" [17], [30] or [32] are well-known for their applicability to distributed synchronisation. Given the limited number of devices in a house, this solution might seem like an overkill. However, synchronisation and coordination issues in smart grids appear as well at the city, region or country level where excellent scalability and robustness are required.

One could also imagine a decentralised solution to the heater management conflict: the two "AP" AMRs (cf. Fig. 9) could either inhibit each other or compute a mean of their plans with a distributed algorithm. However, given the lack of scalability issues, such a solution is arguably overcomplicated for this simple case.

### *G. Meta-Manager pattern*

*1) Context*

The management layer itself needs to be adaptive, self-configuring, self-optimising, self-repairing and self-protecting, in order to cope with versatile and unpredictable application resources and contexts. Quantitative parameters of the AMRs, as well as conflict resolution resources, must be dynamically monitored and adapted throughout the autonomic system's life-cycle.

*2) Description*

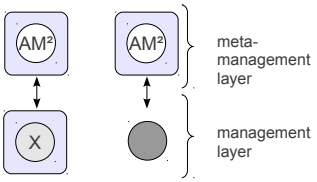


*Figure 16: Meta-manager pattern.*

A Meta-Manager monitors and adapts the conflicting AMR(s), in parallel with the autonomic layer process (upon which it does not intervene directly). Conflicting AMRs are managed application resources for the Meta-Manager.

A Meta-Manager is not a Hierarch since it is a manager of management and conflict resolution resources, whereas the Hierarch is an AMR in the management layer that

performs conflict resolution in the main management flow. The Meta-Manager is not a Controller since a Controller is a special-purpose resource dedicated to conflict resolution, whereas a Meta-manager is a general-purpose, meta-level AMR, here managing AMRs in order to solve conflicts. The Meta-Manager can intervene directly on the AMRs' core business logic, or act to adapt and optimise the AMRs' specific conflict-resolution logic. Finally, the Meta-Manager can also be used to regulate special-purpose conflict-resolution components, such as the Hierarch or the Controller (cf. section H below).

*3) Evaluation*

As the Meta-Manager executes in parallel with the lower-level autonomic management layer, it introduces little or no direct overheads in the autonomic management chains. As a central decision system, it can manage critical properties of the management layer, such as response times or Quality of (management) Service. In particular, the Meta-Manager can adapt and optimise conflict resolution mechanisms available in the management layer. Provided that AMRs are instrumented for meta-management, the management and meta-management layers are clearly decoupled and the Meta-Manager is not intrusive.

Described as such, a centralised Meta-Manager may raise single point of failure and scalability issues. However, since the Meta-Manager does not intervene directly in the management flows or process, its failure will only suppress support for conflict resolution and related adaptations. Finally, the entire meta-management layer can be generalised and implemented to be as rich and diversified as a management layer may be.

*4) Example application within the smart home use case*

The Meta-Manager is not meant to implement one of the conflict resolution mechanisms we described before. However, a Meta-Manager allows making this mechanism adaptive. For instance, supposing an Aggregator is used in the heater conflict case (cf. Aggregator pattern), a Meta-Manager can manage this Aggregator and adapt its semantics (e.g. use either "all or nothing" or "compromise" strategies) according to the context, or to a run-time performance evaluation. One could also imagine a Meta-Manager deploying several resolution patterns successively, and choosing the best one according to metric criteria.

Meta-Managers have been proposed by several software engineering projects in the autonomic domain, e.g. [4,26].

### *H. Pattern combinations*

The Meta-Manager solution is intended to be combined with other conflict resolution mechanisms (or patterns). Actually it is compatible with all previously presented patterns. Conversely, by design, the Monolith is difficult to integrate with any other pattern (except for the Meta-Manager).

The Dealer, Aggregator, Controller, Hierarch and Collaboration patterns can be combined with each other. In particular, a single component can perform both Dealer and Aggregator roles, resulting in a "Sandbox" inside which conflicting AMRs evolve under control, with minimal intrusiveness. The Sandbox is not a Controller, since a Controller features additional communications dedicated to conflict resolution, whereas communications in a Sandbox form a standard management flow.

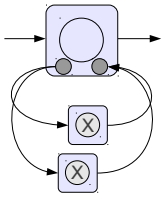

*Figure 17: the Sandbox pattern combination (Dealer + Aggregator).*

Controller and Hierarch mixed together is the kind of hybrid solution that would be particularly appropriate for load management in the smart home use case. Scalable, open, robust but poorly reactive load management is ensured by a decentralised Collaboration that takes into account local appliances objectives, whereas a centralised Hierarch takes arbitrary but quick decisions, forcing designated appliances to shutting down in case of global emergency situations (risk of blackout).

## VI. CONCLUSION AND FUTURE WORK

This paper addressed integration as an essential requirement and challenge in designing, developing and maintaining complex autonomic systems – i.e. systems that are large-scale, dynamic, distributed, open and pursuing multiple, possibly conflicting goals. Such systems were formally modelled based on a highly modular, flexible and open architecture. The paper focused on conflicts as a key integration issue to be addressed in autonomic systems. Different conflict types were identified and depicted based on the formal model, as well as exemplified in the context of a smart home scenario.

The main contribution consists in identifying and specifying a suite of architectural design patterns for solving different classes of integration-related conflicts. The patterns apply to a wide range of situations, depending on the needed conflict resolution type, and they are open to variation and combination. While the example use case shows rather simple scenarios and resolution mechanisms, the patterns are generic enough to embrace a rich spectrum of possibilities. Most of the integration solutions described have already been instantiated in various contexts of autonomic computing, robotics or multi-agent systems (e.g. "Monolith", "Hierarch", "Collaboration" or "Meta-Manager"). The main contribution here is to identify and collect such architectural solutions and propose them as an extensible collection of integration patterns for autonomic computing. Additionally, a number of the presented integration patterns are specific to the architectural model that was adopted for building complex, adaptive autonomic management systems (e.g. "Dealer", "Aggregator" and "Controller").

The patterns were evaluated and compared via a set of quality attributes. While still rather simple and informal, these attributes bring an important evaluation feature that is critical to choosing the appropriate pattern for a given set of requirements. Future efforts will aim at further formalising such quality attributes and gathering concrete experimental data in support of their evaluation.

The example applications of patterns to the smart home use case have been implemented in separate projects, involving various contexts and technologies. This paper's contribution represents a step forward towards providing a unified framework for guiding software engineers and

domain experts from modelling to implementing and to maintaining autonomic management systems. Future work will concentrate on completing this framework and presenting it in the context of the comprehensive use case featuring all presented examples.

## ACKNOWLEDGMENT

This work is supported by EDF R&D via the CIFRE funding of Sylvain Frey's Ph.D. thesis.

## REFERENCES

[1] Jeffrey O. Kephart and David M. Chess. 2003. The Vision of Autonomic Computing. Computer 36, 1 (January 2003), 41-50.

[2] J. W. Sweitzer and C. Draper, Autonomic Computing: Concepts, Infrastructure, and Applications. CRC Press, 2006, ch. 5: Architecture Overview for Autonomic Computing, pp. 71-98.

[3] Y. Maurel, A. Diaconescu and P. Lalanda, "CEYLON : A service-oriented framework for building autonomic managers", 7th IEEE Conference and Workshops on Engineering of Autonomic and Autonomous Systems (EASe) 2010, University of Oxford, England,

[4] Y. Maurel, P. Lalanda and A. Diaconescu, "Towards a service-oriented component model for autonomic management", 8th IEEE International Conference on Services Computing (SCC) 2011.

[5] D.M. Chess, A. Segal, I. Whalley, and S.R. White. Unity : experiences with a prototype autonomic computing system. In Autonomic Computing, 2004.

[6] Shang-Wen Cheng. Rainbow : cost-effective software architecture-based self-adaptation. PhD thesis, Pittsburgh, PA, USA, 2008.

[7] Autonomic Computing Laboratory, University of Arizona. AUTONOMIA: An Autonomic Computing Environment. www2.engr.arizona.edu/~hpdc/projects/AUTONOMIA/

[8] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate : Enabling autonomic applications on the grid. In : autonomic computing workshop, the fifth annual international workshop on active middleware services (AMS) 2003.

[9] Dobson, S.; Sterritt, R.; Nixon, P.; Hinchey, M.; "Fulfilling the Vision of Autonomic Computing," Computer, vol.43, no.1, Jan. 2010.

[10] Bryan Horling and Victor Lesser. 2004. A survey of multi-agent organizational paradigms. Knowl. Eng. Rev. 19, 4 (2004), 281-316.

[11] J. Branke, M. Mnif, C. Müller-Schloer, H. Prothmann, U. Richter, F. Rochner, and H. Schmeck. 2006. Organic Computing - Addressing Complexity by Controlled Self-Organization. In Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISOLA). 2006.

[12] Dobson, Simon. "Facilitating a Well-Founded Approach to Autonomic Systems," Engineering of Autonomic and Autonomous Systems (EASE). 2008.

[13] OSGi alliance. Open Services Gateway initiative. www.osgi.org

[14] Apache Software Foundation. iPOJO, a flexible and extensible service component model. felix.apache.org/site/apache-felix-ipojo.html

[15] Adele team, Laboratoire d'Informatique de Grenoble. Cilia mediation framework. wikiadele.imag.fr/index.php/Cilia

[16] Kung-Kiu Lau and Zheng Wang. 2005. A Taxonomy of Software Component Models. In *Proceedings of the 31st EUROMICRO Conference on Software Engineering and Advanced Applications* (EUROMICRO). 2005.

[17] M. Breza, J. A. McCann. "Can Fireflies Gossip and Flock? The possibility of combining well-known bio-inspired algorithms to manage multiple global parameters in wireless sensor networks without centralised control.", Technical Note, Department of Computing, Imperial College London. 2008.

[18] Cakar, E.; Muller-Schloer, C.; , "Self-Organising Interaction Patterns of Homogeneous and Heterogeneous Multi-Agent Populations," Self-Adaptive and Self-Organizing Systems (SASO). 2009.

[19] O. Babaoglu, G. Canright, A. Deutsch, G. A. Di Caro, F. Ducatelle, Luca M. Gambardella, N. Ganguly, M. Jelasity, R. Montemanni, A. Montresor, and T. Urnes. Design patterns from biology for distributed computing. ACM Trans. Auton. Adapt. Syst. 1, 1 (Sept. 2006), 26-66.

[20] J. L. Fernandez-Marquez, J. Lluis Arcos, G. Di Marzo Serugendo, M. Viroli, and S. Montagna. Description and composition of bio-inspired design patterns: the gradient case. 3rd workshop on Biologically inspired algorithms for distributed systems (BADS). 2011.

[21] Roy T. Fielding. Representational State Transfer (REST) architecture. www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

[22] World Wide Web Consortium (W3C). Web Services Architecture. www.w3.org/TR/ws-arch/

[23] B. Becker, F. Allerding, U. Reiner, M. Kahl, U. Richter, D. Pathmaperuma, H. Schmeck, T. Leibfried. Decentralized Energy-Management to Control Smart-Home Architectures. ARCS'2010.

[24] H. Schmeck, C. Müller-Schloer, E. Çakar, M. Mnif, and U. Richter. Adaptivity and self-organization in organic computing systems. ACM Trans. Auton. Adapt. Syst. 5, 3, Article 10 (2010).

[25] Erann Gat. 1998. Three-layer architectures. In Artificial intelligence and mobile robots, David Kortenkamp, R. Peter Bonasso, and Robin Murphy (Eds.). MIT Press, Cambridge, MA, USA 195-210.

[26] Jeff Kramer and Jeff Magee. Self-Managed Systems: an Architectural Challenge. In 2007 Future of Software Engineering (FOSE) 2007.

[27] Rodney A. Brooks. 1990. Elephants don't play chess. *Robot. Auton. Syst.* 6, 1-2 (June 1990), 3-15.

[28] S. Frey, F. Huguet, C. Mivielle, D. Menga, A. Diaconescu, I. Demeure. "Scenarios for an autonomic micro smart grid". Technical report, CNRS-LTCI Télécom ParisTech. To appear (2011).

[29] Frey, S.; Lalanda, P.; Diaconescu, A.; , "A Decentralised Architecture for Multi-objective Autonomic Management," Self-Adaptive and Self-Organizing Systems (SASO), 2010.

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

[31] M. Jelasity, A. Montresor and O. Babaoglu, "Gossip-based Aggregation in Large Dynamic Networks", ACM Transactions on Computer Systems, 23 (3), pp 219-252, August 2005

[32] R. J. Anthony, "Emergence: a Paradigm for Robust and Scalable Distributed Application", Proceedings of the IEEE International Conference on Autonomic Computing (ICAC), 2004

[33] J. Bourcier, A. Diaconescu, P. Lalanda, J. A. McCann. AutoHome: An Autonomic Management Framework for Pervasive Home Applications. *ACM Trans. Auton. Adapt. Syst.* 6, 1, Article 8. 2011.

[34] IBM. An architectural blueprint for autonomic computing, 4th Edition. IBM Corporation, 2006.

[35] Danny Weyns, Alexander Helleboogh, and Tom Holvoet. 2009. How to get multi-agent systems accepted in industry?; *Int. J. Agent-Oriented Software Engineering* 3, 4 (May 2009), 383-390.

[36] Object Management Group (OMG). CORBA: Common Object Request Broker Architecture. www.corba.org

[37] Oracle. Java EE: Java Platform, Enterprise Edition. www.oracle.com/technetwork/java/javaee

[38] Microsoft. .NET framework. www.microsoft.com/net

[39] Google. Android mobile platform. www.android.com