Modeling Adaptation with Klaim[*]

Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese
Università degli Studi di Firenze
edmondi_gj@yahoo.it, {michele.loreti,rosario.pugliese}@unifi.it

Francesco Tiezzi
IMT Advanced Studies Lucca
francesco.tiezzi@imtlucca.it

ABSTRACT

In recent years, it has been argued that systems and applications, in order to deal with their increasing complexity, should be able to adapt their behavior according to new requirements or environment conditions. In this paper, we present an investigation aiming at studying how coordination languages and formal methods can contribute to a better understanding, implementation and use of the mechanisms and techniques for adaptation currently proposed in the literature. Our study relies on the formal coordination language KLAIM as a common framework for modeling some well-known adaptation techniques: the IBM MAPE-K loop, the Accord component-based framework for architectural adaptation, and the aspect- and context-oriented programming paradigms. We illustrate our approach through a simple example concerning a data repository equipped with an automated cache mechanism.

Categories and Subject Descriptors

C.1.3 [**Other Architecture Styles**]: Adaptable architectures; F.3.1 [**Theory of computation**]: Specifying and Verifying and Reasoning about Programs

Keywords

Autonomic computing, adaptive systems, aspect- and context-oriented programming, coordination languages

1. INTRODUCTION

The increasing scale complexity, heterogeneity and dynamism of networks, systems and applications have made computational and information infrastructure brittle, unmanageable and insecure. This has called for the investigation of an alternate paradigm for designing systems and applications. One popular vision is that of *autonomic computing* [28, 39]: computer and software systems can manage themselves in accordance with high-level guidance from humans by relying on strategies inspired by biological systems.

Autonomic computing encloses the whole spectrum of activities that a system should perform in order to be dynamically and autonomously adaptive. Therefore, an autonomic system should monitor its state and its components, as well as the execution context, and identify relevant changes that may affect the achievement of its goals or the fulfillment of its requirements. The system should then plan reconfigurations in order to meet the new functional or non-functional requirements, execute them, and monitor that its goals are being achieved once again, possibly without any interruption. All these stages make use of a common knowledge that guides the monitoring activities and that may be enriched by the experience earned during execution. The whole body of activities mentioned above has been named *MAPE-K loop* (Monitoring, Analyzing, Planning, and Executing, through the use of Knowledge) by IBM [25].

The key concept of the autonomic computing paradigm is *adaptation*, namely "the capability of a system to change its behavior according to new requirements or environment conditions" [24]. We are interested in the techniques, primitives and mechanisms currently used to achieve the desired adaptation. Indeed, an application can change its behavior in many ways. For example, with a simple *if*(*condition*) statement, an application can choose to change its overall behavior, or part of it. Clearly, this is not a good way to implement adaptation. Authors in [10] argue that it should depend on the semantics we give to *condition* whether the statement can even be considered as an adaptation point or just as an application branch. Anyhow, this adaptation technique is certainly not scalable, nor is it robust or easy to maintain or even comprehend in a complex software system. Much more elaborated techniques are required in real world applications. In the literature, two main approaches have been proposed for implementing adaptation in a software system: *architectural-level* and *language-level*.

The architectural-level approach [38] relies on the runtime structural modification of the software architecture of the system. Typically, this approach is applicable whenever the system is composed of many *components*, possibly interacting through *connectors*, composing thus a *network* which may also be hierarchical and distributed. This is the case, for example, of such component-based programming systems as CORBA Component Model [37] and Common Component Architecture [5]. Adaptation can then be achieved by

[*]This work is based on an earlier work: SAC '12 Proceedings of the 2012 ACM Symposium on Applied Computing, Copyright 2012 ACM 978-1-4503-0857-1/12/03. http://doi.acm.org/10.1145/2245276.2232019.

modifying the way components interact and also by adding or removing components and/or connectors or by replacing them with others. For example, a component may be replaced at run-time by another one that provides a similar basic functionality but with an additional support for a new emerging requirement of a subnet of the system.

The language-level approach extends standard programming languages with primitives and mechanisms that enable to dynamically change the behavior of (part of) a system. In [19] the authors review the adaptation capabilities of traditional programming languages and paradigms. For example, considering object-oriented languages, as today's most used ones for software programming, their analysis shows that class inheritance and method overriding offer some degree of adaptiveness although these mechanisms are not usually dynamic (a notable exception is [8], which presents a Java-like core language using dynamic object composition and 'horizontal' method overriding through delegation). New programming techniques, however, have captured more attention in the years. Until lately, the mainstream techniques have focused on *Aspect-Oriented Programming* (AOP), to enforce the separation of concerns, and on *Dynamic AOP* [21], to support run-time adaptation. More recently, a new promising technique has been specifically proposed for supporting dynamic software adaptation: *context-oriented programming* (COP) [23]. COP uses *ad hoc* explicit language-level abstractions to express context-dependent behavioral variations and their run-time activation.

In this paper, we show how KLAIM [12], a tuple-space-based coordination language, can be used to model adaptive systems. Coordination languages based on tuple spaces have the advantage of providing an accurate model of systems using a small set of primitives and in a clear and accessible way. We have chosen KLAIM as representative of the broad class of coordination languages (see, e.g., [11] for a survey) because it is a formal language coming with software tools supporting both verification and programming. It is also well suited for mobile and distributed applications, characteristics that are today well-consolidated and dominant in the software market.

Specifically, we briefly describe the IBM's MAPE-K loop as well as some relevant techniques using architectural- or language-level approaches to adaptation, such as Accord, AOP and COP, showing also how they can be easily modeled in KLAIM (see Figure 1). For better understanding the mentioned techniques, we use a common, simple example scenario (drawn from [21]), concerning an automated cache system. In this scenario, a *Repository*, containing useful data, can be queried by an application using the provided functionality *request()*. The system may enter the *performance mode*, where a *monitor* checks that the Repository's response time is below a certain threshold T_H. If this threshold is exceeded, a caching system is introduced in the Repository, to reduce the current, possibly large, database access times. If the response time for the cache misses falls under another threshold T_L, the cache is disabled to avoid unnecessary overhead.

We use KLAIM to model each implementation of the automated cache example following the discussed techniques. This way, we get models of different implementations in a common language, which favors comparisons among the modeled techniques. On the basis of this modeling activity,

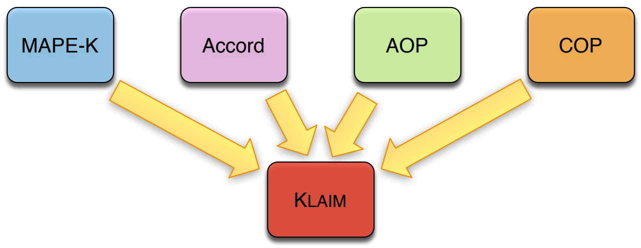


Figure 1: Modeling adaptation techniques in Klaim

we can argue that tuple-based higher-order communication enables a straightforward implementation of dynamic adaptation mechanisms.

Our work paves the way to a twofold application of the know-how in the field of coordination languages and formal methods in the context of adaptive systems. On the one hand, KLAIM formal tools and techniques can be used to support the verification of modeled adaptive systems. Indeed, KLAIM's strong mathematical foundations enable the use of a range of software assisted verification methods, from theorem proving and model checking to simulation and probabilistic analysis (see, e.g., [14, 13]). On the other hand, KLAIM comes with a run-time framework, named KLAVA [7], that permits using KLAIM actions within Java code and, thus, can be used to natively implement adaptive systems.

The rest of the paper is organized as follows. Section 2 provides a brief overview of KLAIM. Sections 3 and 4 present a brief description of the IBM's MAPE-K loop and some architectural-level adaptation techniques and show how they can be rendered in KLAIM, respectively. Section 5 presents the AOP and COP paradigms used for adaptation and autonomic computing and give a flavour of their KLAIM models. Section 6 shows how the different implementations of the automated cache example are modeled in KLAIM. Section 7 presents comparisons with more strictly related work, while Section 8 concludes with some directions for future work.

2. KLAIM

In this section, we summarize the key features of the formal language KLAIM. It has been specifically designed to provide programmers with primitives for handling physical distribution, scoping and mobility of processes. Although KLAIM is based on process algebras, it makes use of Linda-like asynchronous communication and models distribution via multiple shared tuple spaces.

Linda [17] is a coordination paradigm rather than a language, since it only provides a set of coordination primitives. It relies on the so-called *generative communication paradigm*, which decouples the communicating processes both in space and time. Communication is achieved by sharing a common tuple space, where processes insert, read and withdraw tuples. The data retrieving mechanism uses associative pattern-matching to find the required data in the tuple space.

KLAIM enriches Linda primitives with explicit information about the locality where processes and tuples are allocated. KLAIM syntax[1] is shown in Table 1 and Figure 2 is a simple

[1]We use a version of KLAIM enriched with high-level features, such as assignments and standard control flow constructs (i.e., sequence, if-then-else, and while loop), that

Table 1: Klaim syntax

(Nets)	$N \ ::= \ s ::_{\rho} C \ \mid \ N_1 \parallel N_2 \ \mid \ (\nu s)N$
(Components)	$C \ ::= \ P \ \mid \ \langle t \rangle \ \mid \ C_1 \,|\, C_2$
(Processes)	$P \ ::= \ a \ \mid \ X \ \mid \ A(p_1, \ldots, p_n)$
	$\mid \ P_1 \,;P_2 \ \mid \ P_1 \,|\, P_2 \ \mid \ P_1 + P_2$
	$\mid$ **if** (e) **then** $\{P_1\}$ **else** $\{P_2\}$
	$\mid$ **while** (e) $\{P\}$
(Actions)	$a \ ::=$ **in**$(T)@\ell$ $\mid$ **read**$(T)@\ell$
	$\mid$ **inp**$(T)@\ell$ $\mid$ **readp**$(T)@\ell$
	$\mid$ **out**$(t)@\ell$ $\mid$ **eval**$(P)@\ell$
	$\mid$ **newloc**(s) $\mid$ $x \ = \ e$
(Tuples)	$t \ ::= \ e \ \mid \ \ell \ \mid \ P \ \mid \ t_1, t_2$
(Templates)	$T \ ::= e \ \mid \ \ell \ \mid \ P$
	$\mid \ !\,x \ \mid \ !\,l \ \mid \ !\,X \ \mid \ T_1, T_2$

illustration of it.

Nets N are finite plain collections of nodes composed by means of the parallel operator $N_1 \parallel N_2$. It is possible to restrict the scope of a name s by using the operator $(\nu s)N$: in a net of the form $N_1 \parallel (\nu s)N_2$, the effect of the operator is to make s invisible from within N_1.

Nodes $s ::_{\rho} C$ have a unique *locality name* s (i.e. their network address) and an allocation environment ρ, and host a set of components C. The *allocation environment* provides a name resolution mechanism by mapping *locality variables* l (i.e., aliases for addresses), occurring in the processes hosted in the corresponding node, into localities s. The distinguished locality variable **self** is used by processes to refer to the address of their current hosting node. *Components* C are finite plain collections of processes P and evaluated tuples $\langle t \rangle$, composed by means of the parallel operator $C_1 \,|\, C_2$.

Processes P are the KLAIM active computational units, which can be executed concurrently either at the same locality or at different localities. They are built up from basic actions a, process variables X, and process calls $A(p_1, \ldots, p_n)$, by means of sequential composition $P_1; P_2$, parallel composition $P_1 \,|\, P_2$, non-deterministic choice $P_1 + P_2$, conditional choice **if** (e) **then** $\{P_1\}$ **else** $\{P_2\}$, iteration **while** (e) $\{P\}$, and (possibly recursive) process definition $A(f_1, \ldots, f_m) \triangleq P$, where A denotes a process identifier, while f_i and p_j denote formal and actual parameters, respectively. Notably, e ranges over *expressions*, which contain basic values (booleans, integers, strings[2], floats, etc.) and value variables x, and are formed by using the standard operators on basic values and the non-blocking retrieval actions **inp** and **readp** (explained below). In the rest of this section, we will use the notation ℓ to range over locality names s and locality variables l.

simplify the modeling task. Although these features were not included in the original presentation of KLAIM [12], they can be easily rendered with it (by resorting, e.g., to choice, fresh names and recursion in the usual way). The considered language is also equipped with the non-blocking versions of the retrieval actions, i.e. **inp** and **readp**. All the constructs mentioned above are directly supported by the KLAVA framework and other related tools (such as, e.g., X-KLAIM [6] and SAM [34]).

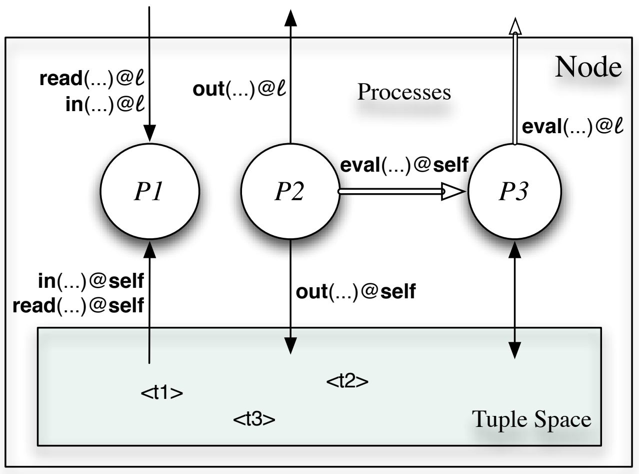


Figure 2: Graphical description of a Klaim node and its components

During their execution, processes perform some basic *actions*. Actions **in**$(T)@\ell$ and **read**$(T)@\ell$ are retrieval actions and permit to withdraw/read data tuples from the tuple space hosted at the (possibly remote) locality ℓ: if a matching tuple is found, one is non-deterministically chosen, otherwise the process is blocked. They exploit templates as patterns to select tuples in shared tuple spaces. *Templates* are sequences of actual and formal fields, where the latter are written $!\,x$, $!\,l$ or $!\,X$ and are used to bind variables to values, locality names or processes, respectively. Actions **inp**$(T)@\ell$ and **readp**$(T)@\ell$ are non-blocking versions of the retrieval actions: namely, during their execution processes are never blocked. Indeed, if a matching tuple is found, **inp** and **readp** act similarly to **in** and **read**, and additionally return the value *true*; otherwise, they return the value *false* and the executing process does not block. **inp**$(T)@\ell$ and **readp**$(T)@\ell$ can be used where either a boolean expression or an action is expected (in the latter case, the returned value is simply ignored). Action **out**$(t)@\ell$ adds the tuple resulting from the evaluation of t to the tuple space of the target node identified by ℓ, while action **eval**$(P)@\ell$ sends the process P for execution to the (possibly remote) node identified by ℓ. Both **out** and **eval** are non-blocking actions. Finally, action **newloc** creates new network nodes, while action $x \ = \ e$ assigns the value of e to x. Differently from all the other actions, these latter two actions are not indexed with an address because they always act locally.

3. AUTONOMIC COMPUTING IN KLAIM

In this section, we present in more detail the architectural blueprint for autonomic computing proposed by IBM and its modeling in KLAIM.

[2]As usual, strings are enclosed within double quotes.

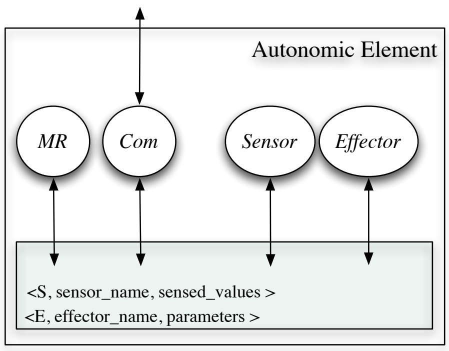

Autonomic Element
MR
Com
Sensor
Effector
<S, sensor_name, sensed_values >
<E, effector_name, parameters >

Figure 3: An autonomic element in Klaim

3.1 An autonomic element

According to the IBM's view, an autonomic element is composed of the managed resource, which is the actual functional (maybe computational, storage, etc.) unit of the system, and the *touchpoint*, which "wraps" the resource by providing a manageability interface to the autonomic manager and other mechanisms implementing the interface's operations. The manageability interface is composed of a sensor and an effector. The sensor exposes information about the current state of a managed resource and may raise an event to capture the attention of the autonomic manager. The effector, instead, enables the manager to change the state of the managed resource, as well as allows the managed resource to make requests to its manager.

In KLAIM, we can model an autonomic element through a node (see Figure 3) as follows:

- the managed resource is rendered in KLAIM by distinguishing between the computational part, i.e. process *MR*, which interacts only with the node's tuple space, and the communication part, i.e. process *Com*, which can interact with other nodes; this distinction is not strictly necessary but it may help the overall system management;
- the *Sensor* process measures relevant parameters in the tuple space and raises events (represented by tuples) caught by the autonomic manager;
- the *Effector* process implements, on the element, the adaptation commands received from the manager.

Some tuples in the tuple space (the S-tagged tuples) carry the sensor's measurements (or the events), others (the E-tagged tuples) describe the installed effectors.

3.2 The MAPE-K loop

The autonomic manager controls the autonomic element through the manageability interface by implementing the MAPE-K loop. In KLAIM, the manager is implemented by a node (see Figure 4). The monitoring phase (i.e. the *Monitor* process) reads the measurements from the sensors and upon recognizing a symptomatic situation it sends the relevant information to the analyze phase. To this aim, the *Monitor* writes "symptom" tuples in the tuple space, which trigger the execution of the *Analyze* process. The symptom is analyzed and, if needed, an adaptation should be performed. An "adaptation request" tuple is produced, specifying what should be adapted, and triggering the execution of the *Plan* process. When the manager decides how should the adaptation be performed, it executes the adaptation on the autonomic element through its effectors. So, the *Plan* process performs an **eval** to launch the appropriate *Execution* process which carries out the planned adaptation.

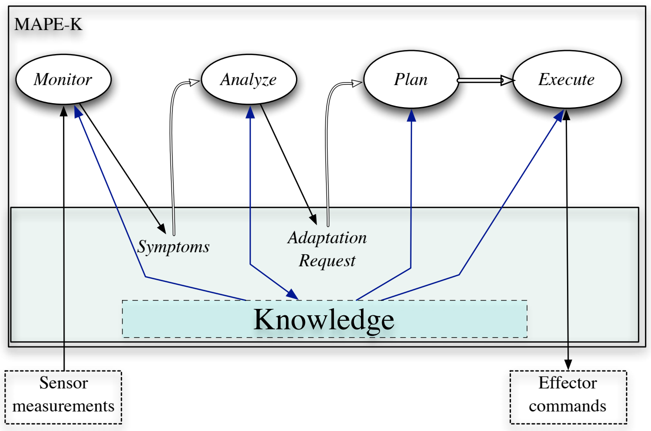


Figure 4: The MAPE-K manager in Klaim

Each of these steps is coordinated by the information stored in the knowledge, represented in KLAIM by a subset of the tuple space. For example, the range of values of a particular measurement considered to be symptomatic is such an information. The analysis phase is the only one that can actually modify the knowledge (notice the bidirectional arrow between the *Analyze* process and the Knowledge). Indeed, this phase is aimed at a wider temporal view of the system, e.g. it may look at the history to see if a symptom occurs too often and thus undertake more drastic (or expensive) adaptations on the system in order to avoid it.

4. MODELING ARCHITECTURAL-LEVEL ADAPTATION

The architectural-level approach relies on dynamic addition/removal/replacement of components and connectors to achieve the desired adaptations. KLAIM makes use of generative communication as in Linda, which is asynchronous, anonymous, public and unaddressed: the communicating processes are completely decoupled from each other. This means that in a set of communicating processes, each process can be replaced (or added or removed) without affecting other processes, as long as this operation maintains the syntax and semantics of the tuples used in the communication. Such property is very advantageous, since it implies that adaptations can be applied to single processes, while any other process is relieved from the duty of taking into account any adaptation external to it, thus better supporting modularity and scalability.

To fully take advantage of this property we could use processes to model each component/connector and model the whole system as a node. However, KLAIM provides better support to model distributed and complex systems when a net with multiple nodes is used to represent the system. This approach can be pursued to provide separation of concerns in the system model, scopes, localities, addressed communication, etc., which is essential for distributed and pervasive systems. In this case, since each node has an explicit locality, we do not have an unaddressed communication and we should take care when adaptation substitutes or deletes entire nodes. Therefore, if, for example, a node A interacts with B, and an adaptation replaces B with C, node A should be notified that it should send messages to C and not to B any longer. Anyway, typically, workarounds to this issue can be easily found (e.g. by using a process in B to forward to C messages coming from A).

In Klaim there are no primitives or clean methods for stopping a process, or asynchronously interrupting its execution when an adaptation has to be performed, and this can be an issue when modelling adaptation at architectural-level. It is still worth noticing that the same problem arises in many other formalisms and languages for concurrent programming (see, e.g., [27] for a discussion on methods to stop Java threads). Some workarounds can be engineered but with some probable issue of effectiveness. A simple way to deal with this drawback is to insert into each Klaim process some "interruption points", where the process checks whether it should continue with the execution of the normal behavior (indicated by the presence of the tuple $\langle pid, \text{“}live\text{”}\rangle$ in the local tuple space, where *pid* is the process identifier) or, otherwise, should stop:

$$\begin{array}{l}\textbf{if } (\textbf{readp}(pid, \text{“}live\text{”})@\textbf{self}) \textbf{ then } \{\ P_{normal}\ \} \\ \textbf{else } \{\ \textbf{out}(pid, \text{“}dead\text{”})@\textbf{self}\ \}\end{array}$$

Before terminating its execution, the process signals this event to its manager by producing the tuple $\langle pid, \text{“}dead\text{”}\rangle$. However, these points must be included by the programmer at design time, which means that we loose some degree of transparency and, since this mechanism is static, it weakens the adaptation capabilities.

Other, more complex, techniques (e.g. designing parametric interactions that can be used to cut out some elements from the system) require typically greater effort. It is easier, when using these techniques, to model the system as a Klaim net (and components as nodes), because we can have a finer-grained control on process execution.

4.1 Accord

Using the above mentioned techniques, we can model, e.g., Accord [33]. Accord is an architectural framework for adaptation and autonomicity that enables the development of autonomic elements and the formulation of autonomic applications as the dynamic composition of autonomic elements. The Accord Autonomic Element (AAE), depicted in Figure 5, is partly inspired by the IBM's proposal. It contains the *computational element*, which is the functional resource, and an *element manager*, which is delegated to manage execution of the computational element by monitoring the state of the element and interacting with other element managers for accomplishing adaptation. Besides, the AAE contains a *functional port*, which describes the functionalities offered and used by the element, a *control port*, which exports sensors and effectors[3] to external managers and controls access to them, and an *operational port*, which is used for the management of execution rules.

Rules are expressed as **if** *"condition" holds* **then** *"action" is executed.* They can be of two types: *behavior rules* and

[3]In [33] the term "actuator" is used instead, but here we use the term "effector", in accordance with IBM's terminology.

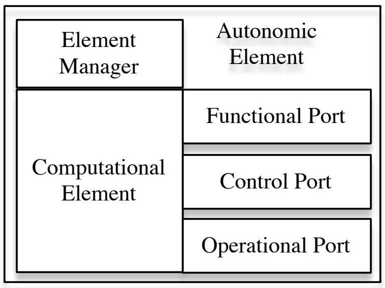


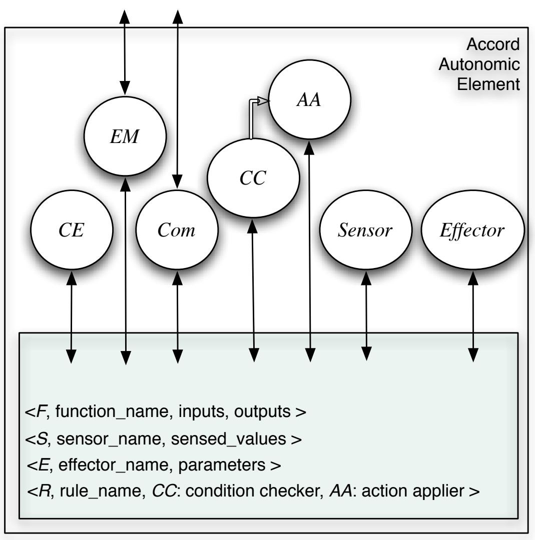
Figure 5: The Accord autonomic element

interaction rules. The former ones control the internal functional behaviors of the current autonomic element, whereas the latter ones control the interaction between the element and the other elements and can modify such interaction accordingly to the verified conditions. Rules execution and management are under the responsibility of the element manager.

Accord implements a sophisticated rule execution mechanism to detect and avoid conflicts, as well as many other features and concepts. These will not be considered here, since they don't fall within the scope of this work. Indeed, our aim is to show that Klaim is expressive enough to conveniently model frameworks for architectural adaptation. Therefore, we treat Accord as a conceptual framework (similarly to the IBM's MAPE-K loop) and will not go into the details of Accord's implementations and their Klaim models.

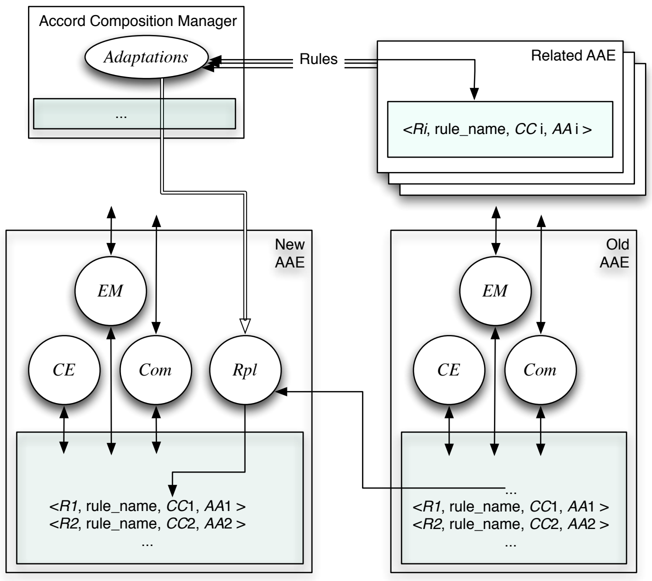
In Klaim, we would model the AAE similarly to what we did with the IBM's autonomic element. In Figure 6, we can see a process *CE* for the computational behavior, which interacts only with the node's tuple space, and a process *Com* for inter-element communication. The element manager as well is a process *EM* that interacts with other elements. The functional port is modeled by a set of F-tagged tuples describing the provided functionalities (for each function, its name, inputs, and outputs are reported). Considering the control port, we use a process *Sensor* to model each sensor and the sensed data (measurements) are rendered as a set of S-tagged tuples. The *Effector* processes may be active, waiting for commands (these are tuples as well), or simply reside in the tuple space, where the manager can get them from and then execute them. The E-tagged tuples can be used for each effector to describe its parameters or the commands it can take. For each rule we have an R-tagged tuple containing two processes. The condition checker process *CC* runs when the rule is executed by the element manager and checks if the condition of the rule is true. If this is the case, it executes the action applier process, *AA*, which applies the rule actions.

As we mentioned at the beginning of this section, the dynamic composition of autonomic elements is the key design choice to implement the autonomic behavior. A unit called *composition manager* (*CM*) is responsible for propagating new interaction rules into related autonomic elements, in order to adapt the structure of the system. Adaptations consist in the addition/removal/replacement of elements, as well as in individual interaction rule adaptation, thus adapting the communication configuration of the system. An existing element can be replaced by another element as long as the functional ports of the two elements are compatible.



Accord Autonomic Element
EM
AA
CC
CE
Com
Sensor
Effector
<F, function_name, inputs, outputs >
<S, sensor_name, sensed_values >
<E, effector_name, parameters >
<R, rule_name, CC: condition checker, AA: action applier >

Figure 6: The Accord element in Klaim



Accord Composition Manager
Adaptations
...
Rules
Related AAE
<Ri, rule_name, CC i, AA i >
New AAE
EM
CE
Com
Rpl
<R1, rule_name, CC1, AA1 >
<R2, rule_name, CC2, AA2 >
...
Old AAE
EM
CE
Com
...
<R1, rule_name, CC1, AA1 >
<R2, rule_name, CC2, AA2 >
...

Figure 7: A Klaim representation of the 'replace' adaptation in Accord

The new element is initiated by the element manager and the old element is notified by the element manager to transit to a stand-by state. The rule set is transferred from the old element to the new one and the interaction rules of related elements are updated, thus establishing the interactions between the new element and those related elements. Addition and deletion of an element is achieved in a similar fashion.

4.1.1 Accord in Klaim

In KLAIM, the composition manager (CM) is a node that directly interacts with any other node (AAE) by sending interaction rules to their tuple spaces (or removing them). By means of such operations, it dynamically revises and reshapes the net structure, thus adjusting the overall system behavior.

In Figure 7, some of the operations performed during the replacement of an autonomic element are depicted. We can see that the composition manager injects the process *Rpl* in the new element. This process copies the set of rules from the old element to the new one and stops the old element. The CM also updates the interaction rules of other AAEs that concern the element-to-replace.

4.1.2 The automated cache in Accord

In the example of the automated cache described in the Introduction, the repository can be implemented as an Accord element that includes the command *request()* in its functional port. When the performance mode is entered, the composition manager inserts a monitor element that checks the response time of the repository, by acting as a forwarder of the request commands and results. If the threshold T_H is exceeded, the manager of the monitor element asks the CM to replace the monitor element with a cache element. The cache captures request commands and, if the requested data is available locally, it immediately responds to the request, otherwise, it sends the request to the repository, also measuring its response time. If the response time falls under T_L, the cache element's manager asks the CM to replace the cache element with a monitor element.

5. MODELING LANGUAGE-LEVEL ADAPTATION

As mentioned in the Introduction, two main language-level adaptation techniques have emerged from recent literature: (dynamic) aspect-oriented programming and context-oriented programming. We now show how they both can be modeled in KLAIM.

It is worth noticing that our aim here is to propose a convenient way of expressing the adaptation mechanisms of the considered languages via a tuple-based formalism, rather than providing an encoding from each language in its completeness into KLAIM. Indeed, to deal with all the functionalities of the considered programming languages, such encodings, on the one hand, would turn out to be too complex and not useful and, on the other hand, would fail on helping the reader to understand and compare the various adaptation approaches.

5.1 Aspect-Oriented Programming

Aspect-oriented programming (AOP) entails breaking down program logic into distinct parts called *concerns*, namely cohesive areas of functionality. Some of these concerns defy traditional forms of encapsulation abstractions (procedures, classes, functions, etc.) and are called *crosscutting concerns* since they "cut across" multiple abstractions in a program.

The AOP paradigm mandates for defining the code of the crosscutting concerns separately from the rest of the application. Without going into too much details, the following are the main concepts at the basis of AOP:

- A *join point* is a point in a running program where additional behavior, from different crosscutting concerns, can be usefully joined. They are typically implicit in the language (e.g. method calls, field read or write accesses, exception handlers).
- A *pointcut* groups a set of join points according to some

of their characteristics (e.g. a pointcut may group all calls of a given method within a given class).

- An *advice* specifies the code to run at a join point. For example, when the join point is a method call, the advice can wrap the called method.

- An *aspect* defines the combination between a pointcut and an advice. The process of applying aspects to a program, and thus producing a program or a program execution modified by the aspects, is called *weaving.*

Adaptation can be seen as a crosscutting concern, since it does not fall within the normal behavior of the application and different adaptations may apply at the same (join) point, depending on the decisions of the adaptation manager. We can identify an adaptation with an aspect. Indeed, this specifies the advice to execute and the pointcut where to execute it.

In an autonomic computing environment, we know that we need to perform dynamic adaptations. *Dynamic* AOP enables to dynamically weave/unweave an aspect into a running application (i.e. the weaving process is performed at runtime without need for recompilation or rebooting). This can be used to program autonomic systems, as shown below in the AOP implementation of the automated cache.

5.1.1 AOP in Klaim

In KLAIM, we can model join points by mandating the application to check, at a given point, if there is any externally defined advice to be performed. Advices are represented by processes that the autonomic manager provides at the aspect weaving. The pointcut is specified by what the program checks for in each join point. Notably, the application only specifies the pointcuts (i.e. the adaptation "hooks"), whereas the advices are defined elsewhere and known only to the autonomic manager. Aspects instead are identified by the autonomic manager which can dynamically weave them on the application.

For example, the following fragment models an application that can be adapted each time process A calls B:

```
$A \triangleq$
  ...
  if (readp("pointcut", "A", "B", !$X_{advice}$)@self) then {
      eval($X_{advice}$)@self
  } else { $B$ }
  ...
```

Above, the autonomic manager can provide an advice P_{advice} for the pointcut by inserting a tuple $\langle$"*pointcut*", "*A*", "*B*", $P_{advice}\rangle$ in the tuple space local to process A. When no such tuple is available, the default behaviour B is executed.

We refer the interested reader to [42] for an extension of KLAIM with primitives and mechanisms for natively dealing with aspects.

5.1.2 JBoss implementation of the automated cache

We implement our reference scenario by using JBoss[4], a Java EE-based application server that, among other features, provides support for aspect-oriented programming and dynamic AOP [26].

[4]JBoss. `http://www.jboss.org`

The pointcut for the request method of the Repository is defined as follows:

```
AdviceBinding requestPointcut =
 new AdviceBinding(
     "execution(
        public String Repository->request(int))",
     null);
```

The object of type AdviceBinding[5] will permit binding the specified pointcut expression to the corresponding advices. This pointcut intercepts all executions of the request method provided by any object of the Repository class.

The Performance aspect is then defined as follows:

```
AspectDefinition performanceAspectDef =
 new AspectDefinition("Performance",
     Scope.PER_INSTANCE,
     new GenericAspectFactory(Performance.class,
                                          null));
```

The definition of an aspect requires to specify its name (e.g. Performance), scope (stating, e.g., that an instance of the aspect will be created for each advised object) and an aspect factory[6] (which creates instances of a given aspect class, e.g. Performance.class, defining the advices). The definition of the performance aspect is then completed by the creation of a factory for the monitor advice (which corresponds to a method defined in the Performance class) and its combination with the pointcut:

```
AdviceFactory monitorFactory =
  new AdviceFactory(performanceAspectDef,
                    "monitor");
requestPointcut.addInterceptorFactory(
                    monitorFactory);
```

This advice will be invoked *around* the joinpoint execution (different types of advice invocation must be explicitly declared).

The Cache aspect is defined in a similar way:

```
AspectDefinition cacheAspectDef =
  new AspectDefinition("Cache",
          Scope.PER_INSTANCE,
          new GenericAspectFactory(Cache.class,
                                      null));
AdviceFactory checkCacheFactory =
  new AdviceFactory(cacheAspectDef,
                    "checkCache");
requestPointcut.addInterceptorFactory(
                    checkCacheFactory);
```

Finally, the performance aspect and the corresponding binding are registered at the AspectManager:

```
AspectManager.instance().
     addAspectDefinition(performanceAspectDef);
AspectManager.instance().
     addAdviceBinding(requestPointcut);
```

Now, when at some point during the execution the request method of a Repository object is called, the call is intercepted and the monitor advice will execute instead:

[5]The second parameter of the AdviceBinding constructor is a control flow string that we do not use in this example.

[6]The second parameter of the GenericAspectFactory constructor is an XML element that allows to pass extra configurations to the aspect class. We do not need such configurations in our example.

```
public Object monitor(@Arg int req){
    long t0=System.currentTimeMillis();
    Object res = CurrentInvocation.proceed();
    long t1=System.currentTimeMillis();
    AutonomicManager autonomicManager =
        AutonomicManager.instance();
    if (! autonomicManager.isCaching())
        autonomicManager.responseTime(t1-t0);
    return res;
}
```

The CurrentInvocation.proceed() statement calls the intercepted method. So, the monitor measures the execution time of the request and calls the *autonomic manager* AutonomicManager to decide what to do. In the autonomic manager we will have:

```
public void reponseTime(long dt){
    AspectManager aspectManager =
           AspectManager.instance();
    if (dt>T_H && !caching){
        aspectManager.removeAspectDefinition(
                 performanceAspectDef);
        aspectManager.addAspectDefinition(
                 cacheAspectDef);
        caching=true;
    }
    if (dt<T_L && caching){
        aspectManager.removeAspectDefinition(
                 cacheAspectDef);
        aspectManager.addAspectDefinition(
                 performanceAspectDef);
        caching=false;
    }
}
```

If needed, i.e. when the threshold T_H is exceeded, the manager dynamically weaves the cache aspect, whereas, if the system was already caching and the response time is below T_L, the aspect is unwoven. The cache aspect intercepts the calls to the request method, and checks and updates the cache:

```
public Object checkCache(@Arg int req){
    Object res=cache.get(req);
    if (res==null){
        long t0=System.currentTimeMillis();
        res = CurrentInvocation.proceed();
        long t1=System.currentTimeMillis();
        cache.put(req,res);
        AutonomicManager autonomicManager =
            AutonomicManager.instance();
        autonomicManager.responseTime(t1-t0);
    }
    return res;
}
```

In case of a cache miss, the intercepted method is called, the result is taken from the repository database and then put into the cache. Moreover, the response time is measured and the autonomic manager is notified.

5.2 Context-Oriented Programming

Context-oriented programming enables the expression of behavioral variations that depend on the context. Context is treated explicitly and the application can dynamically adapt its behavior in response to context changes. The essential linguistic features supporting the COP paradigm are:

- *behavioral variations*: consist of (possibly partial) definitions of behaviors (expressed, e.g., as procedures, functions or methods in the underlying programming model) that can substitute or modify a portion of the basic behavior of the application;
- *layers*: are first-class entities that group related context-dependent behavioral variations;
- *dynamic activation*: the application can decide to activate or deactivate layers dynamically at runtime according to the current context;
- *scoping*: specific constructs can be used to explicitly control the scope within which layers are activated.

Depending on the current execution context, specific layers may be activated and composed at runtime. So, when the application uses the affected functionality, the appropriate variations belonging to the active layers will be executed. In this way, the application's behavior dynamically adapts itself to the current context. Notably, COP does not provide any specific support to model context information, which typically refers to application domain data represented by standard constructs of the underlying programming model.

More layers can be active at a given moment, which may also provide different variations for the same functionality; the mechanism for choosing the variation to execute may depend on the implementation. In this case, variations can also be composed, e.g. a variation from a given layer can call the corresponding variation on the enclosing layer.

When COP is used to implement adaptation in an autonomic computing setting, an autonomic manager can be exploited to recognize a change of context, which then causes the activation of those layers specifically designed to cope with the emerged situation.

5.2.1 COP in Klaim

In KLAIM, layers can be rendered as a set of tuples, each containing the name of the layer, the name of the functionality to be adapted and a process corresponding to the variation. Thus, when the application requires a certain functionality, it simply takes from the tuple space the process corresponding to the variation of the required functionality for the currently active layer, and executes it.

To present in more details how the main features of COP can be expressed in KLAIM, in the rest of this section we refer to the COP language ContextJ [3]. ContextJ is a context-oriented extension of the Java language, where layers are defined within classes and classes thereby carry their own context-specific variations. An example of layers definition in ContextJ is[7]:

```
public class someClass{
  layer l1{
    public static void m1(){...} //variation of m1 in l1
    public static void m2(){...} //variation of m2 in l1
    ...
  }

  layer l2{
    public static void m1(){...} //variation of m1 in l2
    ...
  }
  ...
}
```

[7]For the sake of simplicity, to focus on layers definition and activation while avoiding to deal with objects, we consider here only static void methods without parameters.

The above definition can be rendered in KLAIM as the following set of tuples:

$$\langle\text{“}l1\text{”}, \text{“}m1\text{”}, P_{m1\_l1}\rangle \qquad \langle\text{“}l1\text{”}, \text{“}m2\text{”}, P_{m2\_l1}\rangle$$
$$\langle\text{“}l2\text{”}, \text{“}m1\text{”}, P_{m1\_l2}\rangle \qquad \ldots$$

where P_{mi_lj} represents the code of the variation of method mi within layer lj.

To control scoped layer activation, ContextJ provides a **with** block statement, that can be used as in the following snippet:

```
with (l1){
   m1();
   ...
}
```

Such a method call within a **with** block can be rendered in KLAIM as a process that retrieves a variation process (using the pattern-matching mechanism of KLAIM) and executes it:

$$\begin{array}{l}\textbf{read}(\text{“}l1\text{”}, \text{“}m1\text{”}, !X_{m1\_l1})@\textbf{self};\\ \textbf{eval}(X_{m1\_l1})@\textbf{self};\\ \textbf{in}(\text{“}l1\text{”}, \text{“}m1\text{”}, \text{“}done\text{”})@\textbf{self};\\ \ldots\end{array}$$

Notably, to enable sequential compositions, we assume that each variation process for the method m within the layer l signals its termination by adding a tuple of the form $\langle\text{“}l\text{”}, \text{“}m\text{”}, \text{“}done\text{”}\rangle$ to the tuple space of the hosting node.

In ContextJ, to implement the *dynamic layer combination* [18], which consists of the activation of multiple layers, **with** blocks are simply nested. In this case, if more than one active layer provides a variation for a method, these variations are combined according to the LIFO order: the most recently activated layer is considered first. A variation can also make use of the variations of enclosing layers to accomplish its functionalities. This is achieved through the proceed() method.

To deal with layer combination in KLAIM we should keep track of the layer activation order. We can maintain the LIFO order of layers activation using a queue rendered as a set of tuples:

$$\langle\text{“}active\_layers\text{”}, \text{“}l1\text{”}, \text{“}l2\text{”}\rangle \qquad \langle\text{“}active\_layers\text{”}, \text{“}l2\text{”}, \text{“}l3\text{”}\rangle$$
$$\ldots \qquad \langle\text{“}active\_layers\text{”}, \text{“}lx\text{”}, \text{“}default\text{”}\rangle$$

where a tuple $\langle\text{“}active\_layers\text{”}, \text{“}li\text{”}, \text{“}lj\text{”}\rangle$ indicates that, if li is the layer corresponding to the variation currently considered, then lj is the next layer in the LIFO order.

Statement **with** can in this case be used to add a layer to the LIFO queue. For example, the following ContextJ code

```
with (l1){
    with (l2){
        ...
    }
}
```

can be rendered in KLAIM as follows:

$$\begin{array}{l}\textbf{out}(\text{“}active\_layers\text{”}, \text{“}l1\text{”}, \text{“}default\text{”})@\textbf{self};\\ \textbf{out}(\text{“}active\_layers\text{”}, \text{“}l2\text{”}, \text{“}l1\text{”})@\textbf{self};\\ \ldots\\ \textbf{in}(\text{“}active\_layers\text{”}, \text{“}l2\text{”}, \text{“}l1\text{”})@\textbf{self};\\ \textbf{in}(\text{“}active\_layers\text{”}, \text{“}l1\text{”}, \text{“}default\text{”})@\textbf{self}\end{array}$$

When a variation, e.g. from layer $l2$, performs a proceed(), it executes the variation on layer $l1$. In KLAIM we have:

$$\begin{array}{l}\textbf{read}(\text{“}active\_layers\text{”}, \text{“}l2\text{”}, !x)@\textbf{self};\\ \textbf{read}(x, \text{“}m1\text{”}, !X_{m1\_x})@\textbf{self};\\ \textbf{eval}(X_{m1\_x})@\textbf{self};\\ \textbf{in}(x, \text{“}m1\text{”}, \text{“}done\text{”})@\textbf{self};\\ \ldots\end{array}$$

Notably this variation does not directly refer to the enclosing layer (i.e., $l1$), since in general it might not know statically which layer is activated before $l2$ (see below for dynamic layer activation). Therefore, this technique works for any number of active layers and any proceed() from any layer.

In order to implement autonomic computing in ContextJ, the activation of layers is not specified at compile-time, rather an expression returning the active layer is used as argument of the **with** block, e.g.

```
with (AutonomicManager.instance().getActiveLayer()){
    m1();
    ...
}
```

This is rendered in KLAIM as follows: a process playing the role of the manager maintains up-to-date a tuple $\langle\text{“}active\_layer\text{”}, l\rangle$ containing the name of the active layer, while the KLAIM term modeling the **with** block is

$$\begin{array}{l}\textbf{read}(\text{“}active\_layer\text{”}, !x)@\textbf{self};\\ \textbf{read}(x, \text{“}m1\text{”}, !X_{m1})@\textbf{self};\\ \textbf{eval}(X_{m1})@\textbf{self};\\ \textbf{in}(x, \text{“}m1\text{”}, \text{“}done\text{”})@\textbf{self};\\ \ldots\end{array}$$

To represent layers, rather than using tuples we can alternatively use nodes, and thus locality names. The application reads from its tuple space the locality where to get the variations from. The value of such locality is updated by the autonomic manager, which, by doing so, activates and deactivates layers:

$$\begin{array}{l}\textbf{read}(\text{“}active\_layer\text{”}, !l)@\textbf{self};\\ \textbf{read}(\text{“}m1\text{”}, !X\_m1)@l;\\ \textbf{eval}(X\_m1)@\textbf{self};\\ \textbf{in}(\text{“}m1\text{”}, \text{“}done\text{”})@\textbf{self};\\ \ldots\end{array}$$

5.2.2 ContextJ implementation of the automated cache

We conclude the section by considering again the automated cache example and show a possible implementation using ContextJ. We declare two layers: Performance and Caching:

```
layer Performance{
    public Object request(int req){
        long t0=System.currentTimeMillis();
        Object res=proceed(req);
        long t1=System.currentTimeMillis();
        AutonomicManager autonomicManager =
            AutonomicManager.instance();
        autonomicManager.responseTime(t1-t0);
        return res;
    }
}

layer Caching{
    public Object request(int req){
        Object res=cache.get(req);
        if (res==null){
          long t0=System.currentTimeMillis();
          res=proceed(req);
          long t1=System.currentTimeMillis();
          cache.put(req,res);
          AutonomicManager autonomicManager =
               AutonomicManager.instance();
          autonomicManager.responseTime(t1-t0);
        }
        return res;
     }
}
```

In the autonomic manager, we will have:

```
public void reponseTime(long dt){
     if (dt>T_H && activeLayer!=Caching)
        activeLayer=Caching;
     if (dt<T_L && activeLayer==Caching)
        activeLayer=Performance;
}
```

Initially, the autonomic manager activates the Performance layer. Calls to the request methods are performed in the standard "COP way":

```
with(AutonomicManager.instance().
     getActiveLayer()){
     repository.request(r);
}
```

6. MODELING IMPLEMENTATIONS OF THE AUTOMATED CACHE

In this section, we model in KLAIM the Accord, AOP-JBoss and COP-ContextJ implementations of the automated cache scenario presented in Sections 4 and 5.

Accord

As we have seen in Section 4, the repository, the monitor and the cache are Accord elements that can be added/removed at runtime. A new element will be added by the composition manager upon entering the performance mode. The Performance node will have in its functional port the description for the functionalities request and monitor, as depicted in Figure 8. The composition manager updates interaction rules of other AAEs to perform requests to the Performance node instead of the Repository node. The request function of the Performance node will act as a forwarder to the Repository, and the Performance's element manager will execute the monitor functionality to measure Repository's response time (notice the sensor tuples in the Repository's tuple space).

When the response time exceeds the defined threshold, the Performance's element manager will ask the composition manager to replace it with the Cache element, as we saw in Figure 7 for element replacement. The new scenario will be as what is depicted in Figure 9.

Now, the requests are sent to the Cache node where the bound functionality calls the checkCache function. This function checks the local availability of the required element: if it is found, the result is sent back to the caller; otherwise, the request is forwarded to the Repository, and the element manager will execute the monitor function to check the response time. The result retrieved from the Repository is then stored locally and sent to the caller. If the response time falls under the low threshold, the Cache element manager will request the composition manager to be replaced by the Performance element.

Notably, the three elements repository, monitor and cache, have compatible functional ports, all implementing the request command.

AOP-JBoss

We now model the AOP implementation of the automated cache example. The following two advices express the performance and cache behaviors:

```
$P_{perf}$ ≜
read("Repository","request", !$X_{rep}$)@self;
t0=SystemTime();
eval($X_{rep}$)@self;
in("request result", !res)@self;
t1=SystemTime();
out("time", t1-t0)@self;
out("adviced","request result",res)@self
```

```
$P_{cache}$ ≜
in("request input", !req)@self;
if (not readp(req,!res)@cache) then {
     read("Repository","request",!$X_{rep}$)@self;
     t0=SystemTime();
     eval($X_{rep}$)@self;
     in("request result", !res)@self;
     t1=SystemTime();
     out("time",t1-t0)@self
}
out("adviced","request result",res)@self
```

A process using the repository will execute:

```
if (readp("pointcut","Repository",
          "request",!$X_{advice}$)@self) then {
   eval($X_{advice}$)@self;
   in("adviced","request result",!res)@self; ...
} else {
   read("Repository","request",!$X_{rep}$)@self;
   eval($X_{rep}$)@self;
   in("request result",!res)@self; ...
}
```

To weave the performance aspect, the autonomic manager provides the corresponding advice as follows:

```
out("pointcut","Repository","request",$P_{perf}$)@self
```

The manager will now monitor the response time and weave or unweave the cache aspect by replacing the tuple representing the relative pointcut, according to the repository's response time:

```
in("time",!time)@self;
if (time>T_H and not caching) then {
     caching=true;
     in("pointcut","Repository",
        "request",!X)@self;
     out("pointcut","Repository",
         "request",$P_{cache}$)@self
} else {
     if (time<T_L and caching) then {
          caching=false;
          in("pointcut","Repository",
             "request",!X)@self;
          out("pointcut","Repository",
               "request",$P_{perf}$)@self
     }
}
```

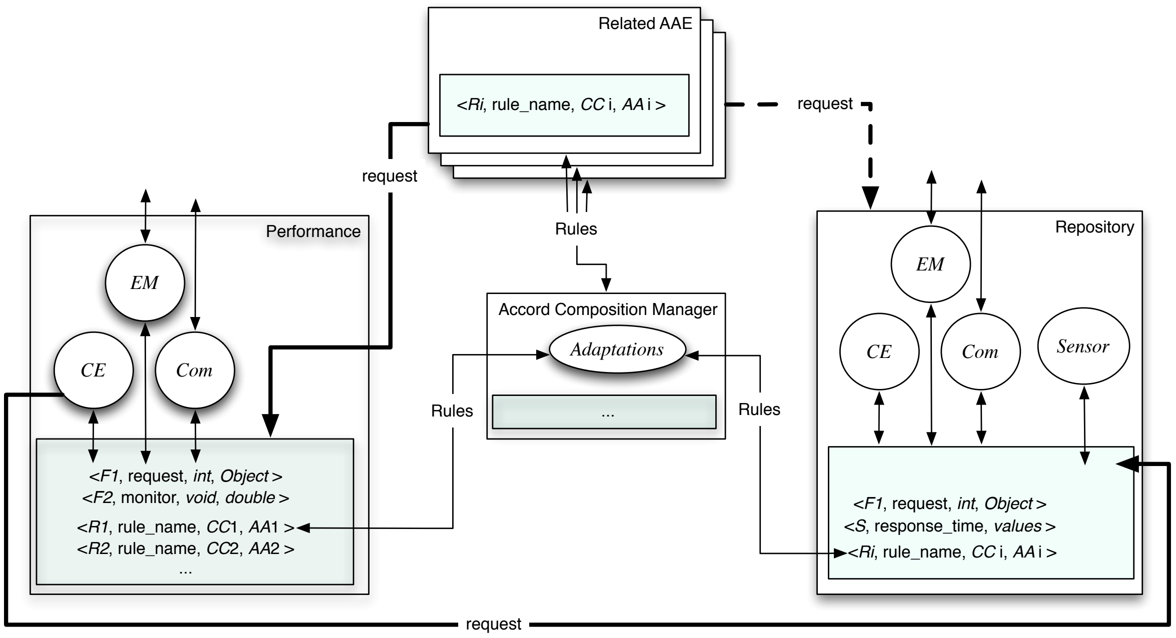

Related AAE
<Ri, rule_name, CC i, AA i >
request
request
Rules
Performance
Repository
EM
CE
Com
Sensor
Accord Composition Manager
Adaptations
...
Rules
Rules
<F1, request, int, Object>
<F2, monitor, void, double>
<R1, rule_name, CC1, AA1 >
<R2, rule_name, CC2, AA2 >
...
<F1, request, int, Object>
<S, response_time, values>
<Ri, rule_name, CC i, AA i >
request

Figure 8: Klaim representation of the Accord implementation of the automated cache (performance mode)

COP-ContextJ

In the Klaim model of the COP implementation, the tuple space of the repository node contains the following tuples representing the two variations of the request method, belonging to the layers Performance and Caching, and the default implementation of the method:

$\langle$"Performance","request",$P_{perf}\rangle$

$\langle$"Caching","request",$P_{cache}\rangle$

$\langle$"Default","request",$P_{default}\rangle$

The processes corresponding to the two variations are similar to the AOP advices previously presented. For example, the variation of the Performance layer is as follows:

```
P_perf ≜
read("Default","request",!X)@self;
t0=SystemTime();
eval(X)@self;
in("Default","request","done",!result)@self;
t1=SystemTime();
out("time",t1−t0)@self;
out("Performance","request","done",result)@self
```

Each time another process needs to retrieve information from the repository, it uses the following instructions:

```
...
read("active_layer",!layer)@self;
read(layer,"request",!X)@self;
eval(X)@self;
in(layer,"request","done",!result)@self;
...
```

where the tuple $\langle$"active_layer",$l\rangle$ is managed by the autonomic manager according to the response time:

```
in("time",!time)@self;
if (time>T_H and not caching) then {
    caching=true;
    in("active_layer",!layer)@self;
    out("active_layer","Caching")@self
} else {
    if (time<T_L and caching) then {
        caching=false;
        in("active_layer",!layer)@self;
        out("active_layer","Performance")@self
    }
}
```

It is worth noticing that, when we pass from the AOP and COP implementations to their models, we take off all the syntactical and language-related features and what we get is the conceptual mechanism implementing the adaptations. As we can see, the two models are very similar, showing that it is the language-related features that make the substantial difference between the two implementations.

7. RELATED WORK

This work starts from [21] and [40, 19], where the authors put forward Dynamic AOP and COP, respectively, as new linguistic techniques that better fit the needs of autonomic computing. Furthermore, several recent works on the COP paradigm have been proposed, primarily presenting implementations that try to best meet the self-adapting requirements. In [20] and [36], COP implementations for mobile applications (for iPhone) are shown. [35] applies the COP paradigm to architectural-level adaptation techniques and shows how Context-Oriented Component-based Applications (COCA) are well suited for developing self-adaptive context-oriented software. The COCA middleware is a framework for designing the architecture of adaptive applications according to a proposed design pattern. In our paper, we take a wider view at autonomicity and adaptivity based on approaches commonly found in the literature, and use the coordination language Klaim to model a reference scenario according to the styles put forward by the different

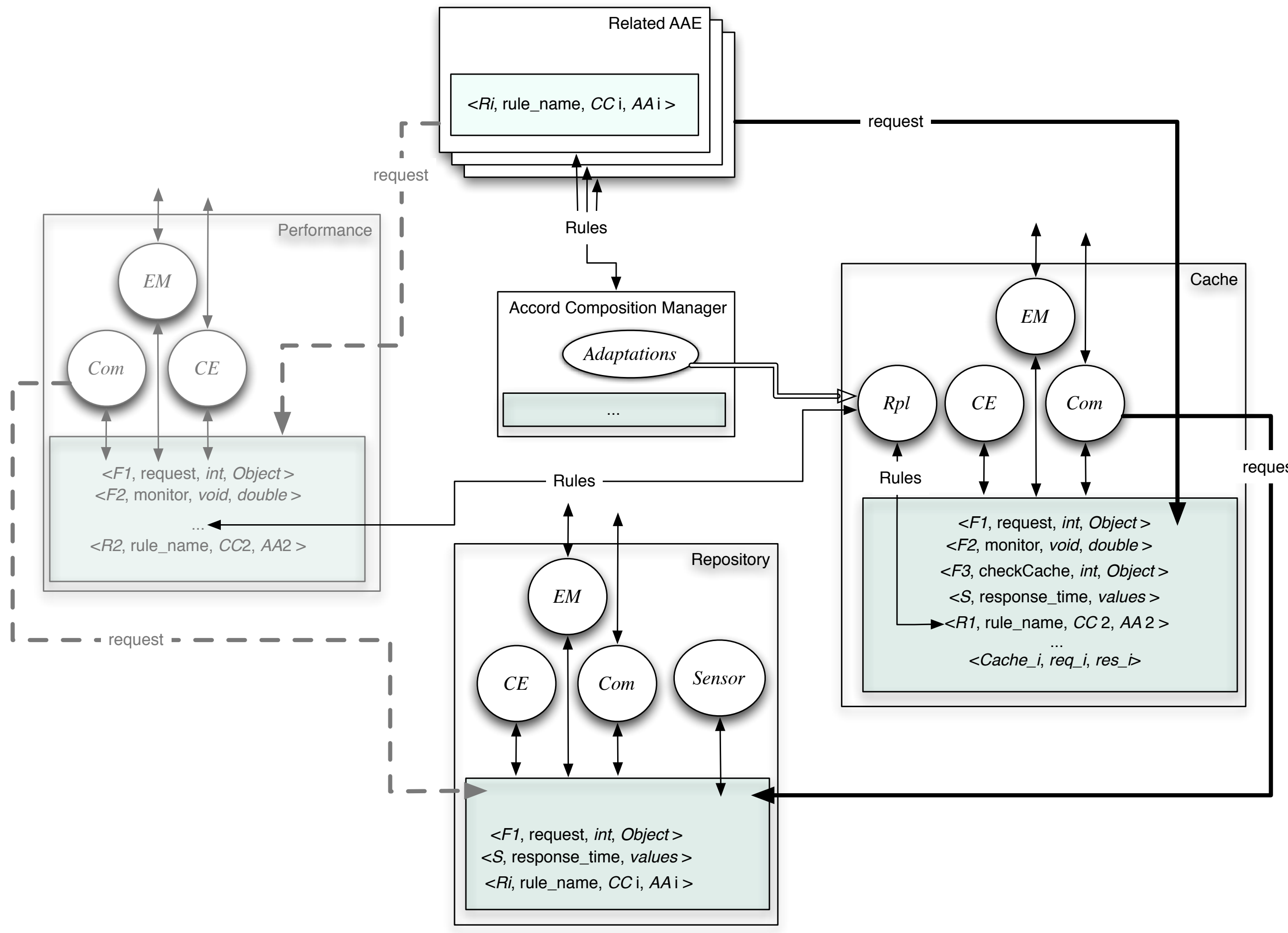

Related AAE
<Ri, rule_name, CC i, AA i >
request
request
Rules
Performance
EM
Com
CE
<F1, request, int, Object>
<F2, monitor, void, double>
...
<R2, rule_name, CC2, AA2 >
Accord Composition Manager
Adaptations
...
Cache
EM
Rpl
CE
Com
Rules
request
<F1, request, int, Object>
<F2, monitor, void, double>
<F3, checkCache, int, Object>
<S, response_time, values>
<R1, rule_name, CC 2, AA 2 >
...
<Cache_i, req_i, res_i>
Rules
request
Repository
EM
CE
Com
Sensor
<F1, request, int, Object>
<S, response_time, values>
<Ri, rule_name, CC i, AA i >

Figure 9: Klaim representation of the Accord implementation of the automated cache (caching mode)

approaches.

As representative of the architectural-level approach to adaptation, we have considered Accord. However, other frameworks for architectural adaptation are surveyed in [22], such as the dynamic configuration support of CONIC [30] and the Rainbow framework [16]. Anyway, at conceptual level, for most of them adaptation means addition/removal/replacement of components and connectors. Hence, the underlying adaptation mechanisms can be modelled in Klaim by following the approach described in Section 4.

Several works have been proposed that use formal methods to model autonomic computing techniques. For example, [9] presents an approach to develop an autonomic service-oriented architecture. However, this and other examples (e.g., [32, 15]) focus on the use of formal methods for specific target applications. Our work, instead, aims at modeling general techniques commonly used to achieve autonomicity rather than specific autonomic systems.

Other coordination languages have been considered for implementing autonomic features. For example, [1] proposes the language ASSIST. This language is very specific for grid computing. On the contrary, Klaim is suitable for modeling and programming any distributed system. Moreover, being based on formal methods, Klaim enables several verifications techniques. As another example, [4] uses the Gamma formalism, a computing model inspired by the chemical reaction metaphor, to develop a higher-order coordination language for specifying autonomic systems. Similarly, [2] presents a biochemical calculus expressive enough to represent adaptive systems, together with a formal framework for property checking. [41] also present a framework of self-organizing coordination with notions of context-dependency, inspired by pervasive ecosystems. The ecosystem is virtually represented by a network of Live Semantic Annotations and ruled by a set of ecolaws, based on semantic chemistry. Differently from the above mentioned works, we consider more systematically the various approaches found in the literature, showing how Klaim can be used to model them.

8. CONCLUDING REMARKS

In the coordination community many languages and formal tools have been proposed to support development and analysis of concurrent and distributed systems. One of these language is Klaim, a successful tuple-space-based coordination language coming with verification tools and techniques and with a full-fledged implementation [6].

In this paper, we have modelled some commonly used adaptation techniques with Klaim. Our work shows that adaptive behaviors can be easily rendered in a coordination language with tuple-based, higher-order communication and that these features enable a straightforward implementation of dynamic adaptation. This is a first step towards a comprehensive study of the relationship between coordination languages and adaptation approaches.

As a future work, we plan to consider further approaches to adaptation (like the rule-based one, see e.g. [31], or the policy-based one, see e.g. [29]) and, at the same time, to provide a formal proof of relative expressiveness of the primitives suggested by the considered approaches. In particular, we intend to assess our approach by studying the relative expressive power of (plain) KLAIM w.r.t. some of its extensions equipped with different adaptation primitives. This study aims at demonstrating that, although these primitives provide full support to a more painless and trouble-free development of autonomic applications, they do not add expressive power to KLAIM, therefore its linguistic constructs are already enough powerful for modeling adaptive behaviors. Finally, we also intend to consider other 'traditional' languages (e.g., Java) and extend them with tuple-based higher-order communication in order to enable the implementation of dynamic adaptations.

Acknowledgments

This work has been partially sponsored by the EU project ASCENS (257414).

9. REFERENCES

[1] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST Grid-Aware Components. In *PDP*, pages 221–230. IEEE, 2006.

[2] O. Andrei and H. Kirchner. A Higher-Order Graph Calculus for Autonomic Computing. In *Graph Theory, Computational Intelligence and Thought*, pages 15–26. Springer, 2009.

[3] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: Context-oriented programming with java. *Computer Software*, 28(1):272–292, 2011.

[4] J.-P. Banâtre, Y. Radenac, and P. Fradet. Chemical Specification of Autonomic Systems. In *IASSE*, pages 72–79. ISCA, 2004.

[5] Bernholdt, D. E., et al. A Component Architecture for High-Performance Scientific Computing. *Int. J. High Perform. Comput. Appl.*, 20(2):163–202, 2006.

[6] L. Bettini, R. De Nicola, G. L. Ferrari, and R. Pugliese. Interactive Mobile Agents in X-Klaim. In *WETICE*, pages 110–117. IEEE, 1998.

[7] L. Bettini, R. De Nicola, and R. Pugliese. Klava: a Java Package for Distributed and Mobile Applications. *Software - Practice and Experience*, 32:1365–1394, 2002.

[8] L. Bettini and B. Venneri. Object reuse and behavior adaptation in java-like languages. In *PPPJ*, pages 111–120. ACM, 2011.

[9] M. A. C. Bhakti and A. Azween. Formal modeling of an autonomic service oriented architecture. In *CSIT*, volume 5, pages 23–29. IACSIT Press, 2011.

[10] R. Bruni, A. Corradini, F. Gadducci, A. Lluch Lafuente, and A. Vandin. A conceptual framework for adaptation. In *FASE*, volume 7212 of *LNCS*, pages 240–254. Springer, 2012.

[11] P. Ciancarini and T. Kielmann. Coordination models and languages for parallel programming. In *PARCO*, pages 3–17. Imperial College Press, 1999.

[12] R. De Nicola, G. Ferrari, and R. Pugliese. Klaim: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[13] R. De Nicola, J. P. Katoen, D. Latella, M. Loreti, and M. Massink. Model checking mobile stochastic logic. *Theor. Comput. Sci.*, 382(1):42–70, 2007.

[14] R. De Nicola and M. Loreti. A modal logic for mobile agents. *ACM Trans. Comput. Log.*, 5(1):79–128, 2004.

[15] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *IPCCC*, pages 61–68. IEEE, 2003.

[16] D. Garlan, S. Cheng, A. Huang, B. Schmerl, and P. Steenkiste. Rainbow: Architecture-based self-adaptation with reusable infrastructure. *IEEE Computer*, 37(10):46–54, 2004.

[17] D. Gelernter. Generative communication in Linda. *ACM Trans. Program. Lang. Syst.*, 7:80–112, 1985.

[18] C. Ghezzi, M. Pradella, and G. Salvaneschi. Programming Language Support to Context-Aware Adaptation—A Case-Study with Erlang. In *SEAMS*, pages 59–68. ACM, 2010.

[19] C. Ghezzi, M. Pradella, and G. Salvaneschi. An Evaluation of the Adaptation Capabilities in Programming Languages. In *SEAMS*, pages 50–59. ACM, 2011.

[20] S. González, N. Cardozo, K. Mens, A. Cádiz, J.-C. Libbrecht, and J. Goffaux. Subjective-C: bringing context to mobile platform programming. In *SLE*, volume 6563 of *LNCS*, pages 246–265. Springer, 2011.

[21] P. Greenwood and L. Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. In *DAW,* RIACS Technical Report 04.01, pages 76–88, 2004.

[22] Hadaytullah. A short survey on self-architecting software systems. Technical report, Tampere University of Technology, Finland. Available at `http://www.cs.tut.fi/~hadaytul/Download/1_Hadaytullah_Survey3.pdf`.

[23] R. Hirschfeld, P. Costanza, and O. Nierstrasz. Context-oriented programming. *Journal of Object Technology*, 7(3):125–151, 2008.

[24] M. Hölzl, A. Rauschmayer, and M. Wirsing. Software engineering for ensembles. In *Software-Intensive Systems and New Computing Paradigms*, pages 45–63. Springer, 2008.

[25] IBM. An architectural blueprint for autonomic computing. Technical report, June 2005. Third edition.

[26] N. Janssens, E. Truyen, F. Sanen, and W. Joosen. Adding dynamic reconfiguration support to JBoss AOP. In *MAI*, pages 1–8. ACM, 2007.

[27] Java 2 Platform Standard Edition 5.0 guide. Why are thread.stop, thread.suspend, thread.resume and runtime.runfinalizersonexit deprecated? Technical report, Oracle, 2010.

[28] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *IEEE Computer*, 36:41–50,

2003.

[29] N. Khakpour, S. Jalili, C. Talcott, M. Sirjani, and M. Mousavi. Formal modeling of evolving self-adaptive systems. *Science of Computer Programming*, 78(1):3 – 26, 2012.

[30] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Trans. Software Eng.*, 11(4):424–436, 1985.

[31] I. Lanese, A. Bucchiarone, and F. Montesi. A framework for rule-based dynamic adaptation. In *TGC*, volume 6084 of *LNCS*, pages 284–300. Springer, 2010.

[32] Z. Li and M. Parashar. Rudder: An agent-based infrastructure for autonomic composition of grid applications. *Multiagent and Grid Systems*, 1(3):183–195, 2005.

[33] H. Liu and M. Parshar. Accord: A programming framework for autonomic applications. *IEEE Trans. on Systems, Man and Cybernetics - PartC: Applications and Reviews*, 36(3):341–353, 2006.

[34] M. Loreti. SAM: Stochastic Analyser for Mobility, 2010. Available at `http://rap.dsi.unifi.it/SAM/`.

[35] B. Magableh and S. Barrett. Adaptive Context Oriented Component-Based Application Middleware (COCA-Middleware). In *UIC*, volume 6905 of *LNCS*, pages 137–151. Springer, 2011.

[36] B. Magableh and S. Barrett. Objective-cop: Objective context oriented programming. In *ICICS*, pages 45–49, 2011.

[37] OMG. CORBA Component Model Specification Version 4.0. Technical report, April 2006.

[38] P. Oreizy et al. An Architecture-Based Approach to Self-Adaptive Software. *IEEE Intelligent Systems*, 14:54–62, 1999.

[39] M. Parashar and S. Hariri. Autonomic Computing: An Overview. In *UPP*, volume 3566 of *LNCS*, pages 257–269. Springer, 2005.

[40] G. Salvaneschi, C. Ghezzi, and M. Pradella. Context-oriented programming: A programming paradigm for autonomic systems. *CoRR*, abs/1105.0069, 2011.

[41] M. Viroli, D. Pianini, S. Montagna, and G. Stevenson. Pervasive Ecosystems: a Coordination Model Based on Semantic Chemistry. In *SAC*, pages 295–302. ACM, 2012.

[42] F. Yang, T. Aotani, H. Masuhara, F. Nielson, and H. R. Nielson. Combining Static Analysis and Runtime Checking in Security Aspects for Distributed Tuple Spaces. In *COORDINATION*, volume 6721 of *LNCS*, pages 202–218. Springer, 2011.