Modeling adaptation with a tuple-based coordination language*

Edmond Gjondrekaj, Michele Loreti, Rosario Pugliese
Università degli Studi di Firenze
gjondrek@dsi.unifi.it, {loreti,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 a preliminary investigation aiming at studying how coordination languages and formal methods can contribute to a better understanding, implementation and usage 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 adaptation techniques, namely the MAPE-K loop, aspect- and context-oriented programming.

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* [21, 25]: computer and software systems can manage themselves in accordance with high-level guidance from humans by relying on strategies inspired by biological systems.

*This work has been partially sponsored by the EU project ASCENS (257414) and by MIUR (PRIN 2009 DISCO).

SAC'12 March 25-29, 2012, Riva del Garda, Italy.

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 [20].

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" [19]. 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 [24] 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. Adaptation is then achieved by 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 [16] 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 the mechanism usually is not dynamic (a notable exception is [6], 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

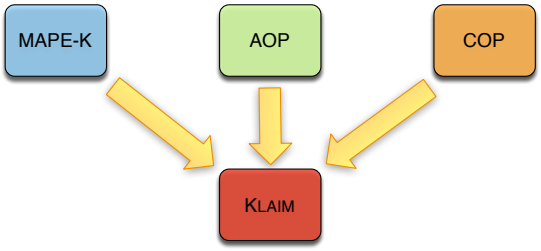

Figure 1: Modeling adaptation techniques in Klaim

have focused on *Aspect-Oriented Programming* (AOP), to enforce the separation of concerns, and on *Dynamic AOP*, to support run-time adaptation [18]. More recently, a new promising technique has been specifically proposed for supporting dynamic software adaptation: *context-oriented programming* (COP) [26]. 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 a tuple-space-based coordination language, namely KLAIM [10], 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. KLAIM, in particular, is well suited for mobile and distributed applications, characteristics that are today well-consolidated and dominant in the software market.

Due to lack of space, here we focus on the language-level approach to adaptation and refer the interested reader to [17] for a more complete account. Specifically, we show how the generic MAPE-K loop, and the techniques AOP and COP can be easily modeled in KLAIM (see Figure 1).

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 wide range of software assisted verification methods, from theorem proving and model checking to simulation and probabilistic analysis (see, e.g., [12, 11]). On the other hand, the programming language derived from KLAIM, namely X-KLAIM [5], can be used to implement such adaptive systems.

The rest of the paper is organized as follows. Section 2 provides a brief overview of KLAIM. Sections 3 and 4 show how the IBM's MAPE-K loop and some language-level adaptation techniques can be rendered in KLAIM, respectively. Section 5 touches upon comparisons with related work. Section 6 concludes with some directions for future work.

2. KLAIM

In this section, we summarize the key features of KLAIM, a formal language that we have chosen as representative of the broader class of coordination languages (see, e.g., [9] for a survey). KLAIM 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 [14] is a coordination paradigm rather than a language, since it only provides a set of coordination primitives.

(*Nets*)	$N ::= \mathbf{0} \mid s ::_\rho C \mid N_1 \parallel N_2 \mid (\nu s)N$
(*Components*)	$C ::= P \mid \langle t \rangle \mid C_1 \mid C_2$
(*Processes*)	$P ::= \mathbf{nil} \mid a.P \mid P_1 \mid P_2 \mid P_1 + P_2 \mid A(\bar{p})$
(*Actions*)	$a ::= \mathbf{out}(t)@\ell \mid \mathbf{in}(T)@\ell \mid \mathbf{read}(T)@\ell$ $\mid \mathbf{eval}(P)@\ell \mid \mathbf{newloc}(s)$
(*Tuples*)	$t ::= e \mid \ell \mid P \mid t_1, t_2$
(*Templates*)	$T ::= e \mid \ell \mid P \mid !\,x \mid !\,l \mid !\,X \mid T_1, T_2$

Table 1: Klaim syntax

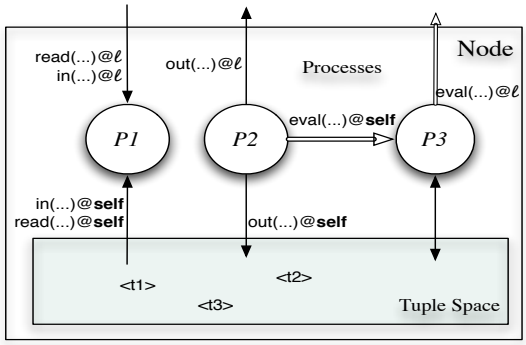

Figure 2: Graphical description of a Klaim node and its components

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 depicted in Figure 2.

Nets are finite plain collections of nodes where components, i.e. processes and evaluated tuples, can be allocated. It is possible to restrict the scope of a name s by using the operator $(\nu s)\_$: 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 have a unique locality s (i.e. their address) and an allocation environment ρ, and host a set of components C. An allocation environment provides a name resolution mechanism by mapping locality variables l, 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.

Processes are the KLAIM active computational units. They are built up from the special process **nil**, which does not perform any action, and from the basic actions by means of action prefixing $a.P$, parallel composition $P_1 \mid P_2$, nondeterministic choice $P_1 + P_2$ and process definition. Processes may be executed concurrently either at the same locality or at different localities and can use the distinguished

[1]We use here the syntax introduced in [7] rather than that of the original paper [10].

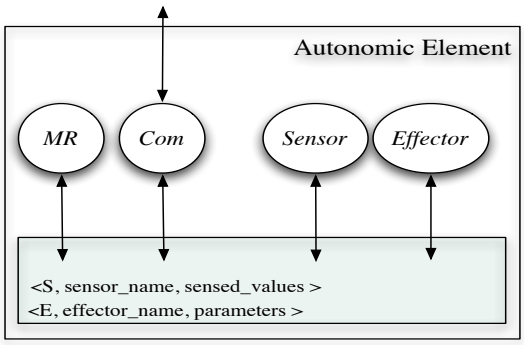

Figure 3: An autonomic element in Klaim

locality variable **self** to refer to the address of their current hosting node. They can perform five different basic actions: the Linda actions **out**, **in** and **read** to insert, withdraw and read tuples, action **eval** to send a process for execution to a (possible remote) node, and action **newloc** to create a new locality (i.e. network node). Actions **in** and **read** are blocking and exploit templates as patterns to select data in shared repositories. A template is a sequence of actual and formal fields, the latter, written as $!x$, $!l$ or $!X$, being used to bind variables to values, locality names or processes, respectively. Notationally, e will range over expressions, whose exact syntax is intentionally left unspecified, while ℓ ranges over locality names s and locality variables l.

3. AUTONOMIC COMPUTING IN KLAIM

In this section, we first enter a little more in the details of the architectural blueprint for autonomic computing proposed by IBM, then we present our modeling in KLAIM.

Autonomic element

According to the IBM's view, an autonomic element is composed of the managed resource, which is the actual functional (may be computational, storage, etc.) unit of the system, and the *touchpoint*, that "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 (with other nodes), i.e. process *Com* (this is not strictly necessary, but we think it may help the system management);
- the sensor process *Sensor* measures relevant parameters in the tuple space and raises events (represented by tuples) caught by the autonomic manager;
- the effector process *Effector* implements, on the element, the adaptation commands received from the manager.

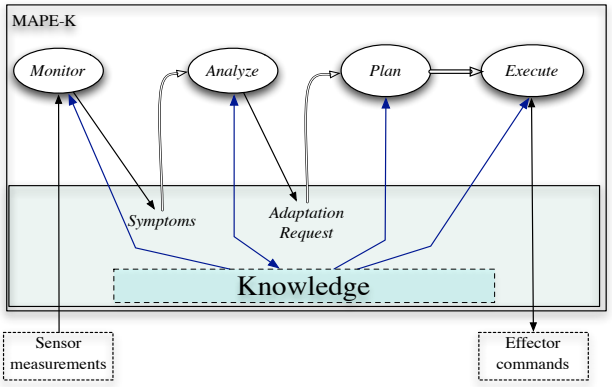

Figure 4: The MAPE-K manager in Klaim

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.

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 sensor 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 effector. So, the *Plan* process performs an **eval** to launch the appropriate *Execution* process which carries out the planned adaptation.

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 analyze phase is the only one that can actually modify the knowledge (notice the bidirectional arrow between the *Analyze* process and the Knowledge), since it is aimed at a vaster temporal view of the system, e.g. it may look in 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 LANGUAGE-LEVEL ADAPTATION WITH KLAIM

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

4.1 Aspect-Oriented Programming

Aspect-oriented programming 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.

Aspect-oriented programming is a paradigm that mandates for defining the code of the crosscutting concerns separately from the rest of the application. Without wanting to enter in too much details, the following are the main concepts used in 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 access, 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.

Adaptation can be seen as a crosscutting concern, since it doesn't 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, then, identify an adaptation with an aspect, since 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 an aspect into a running application. This can be used to implement autonomic systems, as shown in [18].

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 by the autonomic manager. Aspects instead are defined by the 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:

$$
\begin{array}{ll}
A \triangleq & \ldots \\
& \mathbf{read}(\text{“}\mathit{pointcut}\text{”}, \text{“}\mathit{call}\text{”}, \text{“}B\text{”}, \text{“}\mathit{caller}\text{”}, \text{“}A\text{”}, \\
& \qquad !X\_\mathit{advice})@\mathbf{self}.\ \mathbf{eval}(X\_\mathit{advice})@\mathbf{self} \\
& + \mathbf{read}(\text{“}\mathit{no\_pointcut}\text{”}, \text{“}\mathit{call}\text{”}, \text{“}B\text{”}, \text{“}\mathit{caller}\text{”}, \\
& \qquad \text{“}A\text{”})@\mathbf{self}.\ B \\
& \ldots
\end{array}
$$

Above, the autonomic manager can provide an advice P_advice for the pointcut by inserting a tuple $\langle$“*pointcut*”, “*call*”, “B”, “*caller*”, “A”, $P\_advice\rangle$ in the tuple space local to process A, while it can provide no advice by means of tuple $\langle$“*no_pointcut*”, “*call*”, “B”, “*caller*”, “A”$\rangle$.

We refer the interested reader to [27] for an extension of KLAIM with primitives and mechanisms for directly dealing with aspects.

4.2 Context-Oriented Programming

Context-oriented programming enables the expression of behavioral variations dependent on the context. Context is treated explicitly and the application can dynamically adapt its behavior in response to context changes. The essential language features to support 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 with standard constructs of the underlying programming model.

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

In KLAIM, a layer 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.

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.

We present below how the main features of COP can be expressed in KLAIM, by relying on ContextJ [3] as reference COP language. ContextJ is a context-oriented extension to the Java language, where layers are defined within classes, and classes thereby carry their own context-specific variations. An example of layers definition is:

```
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
    ...
  } ...
}
```

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 e.g. as follows:

$$\textbf{with}\ (l1)\{\ \ m1();\ \ldots\ \}$$

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:

$$\textbf{read}(\text{“}l1\text{”}, \text{“}m1\text{”}, !X_{m1\_l1})@\textbf{self}.$$
$$\textbf{eval}(X_{m1\_l1})@\textbf{self}.$$
$$\textbf{in}(\text{“}l1\text{”}, \text{“}m1\text{”}, \text{“}done\text{”})@\textbf{self}.$$
$$\ldots$$

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* [15], 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 their functionalities, and this is achieved through the `proceed()` method. Moreover, `before` and `after` modifiers can be used to execute a behavior before or after a given method execution.

To deal with such features in KLAIM we should keep track of the layer activation. We can maintain the LIFO order of layer activations using a queue modeled by the tuples:

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

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

The **with** statement, in this case, adds a layer to the LIFO queue. For example, the following ContextJ code

$$\textbf{with}\ (l1)\{\ \ \textbf{with}\ (l2)\{\ \ldots\ \}\ \}$$

can be rendered in KLAIM as follows:

$$\textbf{out}(\text{“}active\_layers\text{”}, \text{“}l1\text{”}, \text{“}default\text{”})@\textbf{self}.$$
$$\textbf{out}(\text{“}active\_layers\text{”}, \text{“}l2\text{”}, \text{“}l1\text{”})@\textbf{self}.$$
$$\ldots$$
$$\textbf{in}(\text{“}active\_layers\text{”}, \text{“}l2\text{”}, \text{“}l1\text{”})@\textbf{self}.$$
$$\textbf{in}(\text{“}active\_layers\text{”}, \text{“}l1\text{”}, \text{“}default\text{”})@\textbf{self}$$

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

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

Notably this variation does not directly refer to the enclosing layer ($l1$), 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.

$$\textbf{with}\ (AdaptationManager.getLayer())\{\ \ m1();\ \ldots\ \}$$

This is rendered in KLAIM as follows: a process playing the role of the adaptation 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

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

We conclude the section by discussing an alternative way to represent layers. Rather than using tuples to model layers, we can 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:

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

5. RELATED WORK

Our starting points are [26] and [16], where the authors put forward Context-Oriented Programming as a new linguistic technique that better fits the needs of autonomic computing. In this paper, we briefly present a wider view at autonomicity and adaptivity based on common approaches found in literature, and we use the coordination language KLAIM to model some examples.

Other coordination languages have been considered for implementing autonomic features. For example, [1] proposes the language ASSIST. This language is however very specialized for grid computing, instead 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. Differently from the above mentioned works, we consider more systematically the various approaches found in literature and show how KLAIM can be used to model them (for more details, please refer to [17]).

Several works have been proposed that use formal methods to model autonomic computing techniques. For example, [8] presents an approach to develop an autonomic service-oriented architecture. This and other examples (e.g.,

[23, 13]), however, 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.

6. 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 this language is KLAIM, a successful tuple-space-based coordination language coming with verification tools and techniques and with a full-fledged implementation [5].

In this paper, we have shown that adaptive behaviors can be easily rendered in a coordination language by modeling some popular adaptation techniques with KLAIM. We point out that tuple-based higher-order communication enables a straightforward implementation of dynamic adaptation. This work 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. [22]) and, at the same time, to provide a formal proof of relative expressiveness of the primitives provided by the considered approaches. In particular, we intend to provide some assessments of 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 these primitives do not add expressive power to KLAIM and that its features are enough for modeling adaptive behaviors. Finally, we also intend to consider other traditional languages and extend them with tuple-based higher-order communication in order to enable the implementation of dynamic adaptations.

7. REFERENCES

[1] M. Aldinucci, M. Danelutto, and M. Vanneschi. Autonomic QoS in ASSIST Grid-Aware Components. In *PDP*, pp. 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*, *LNCS* 5420, pp. 15–26. Springer, 2009.

[3] M. Appeltauer, R. Hirschfeld, M. Haupt, and H. Masuhara. Contextj: context-oriented programming with Java. *JSSST Journal*, 2011. To appear.

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

[5] L. Bettini, R. D. Nicola, G. L. Ferrari, and R. Pugliese. Interactive mobile agents in X-Klaim. In *WETICE*, pp. 110–117. IEEE, 1998.

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

[7] L. Bettini et al. The Klaim Project: Theory and Practice. In *Global Computing*, *LNCS* 2874, pp. 88–150. Springer, 2003.

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

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

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

[11] 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.

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

[13] X. Dong et al. Autonomia: an autonomic computing environment. In *IPCCC*, pp. 61–68. IEEE, 2003.

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

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

[16] C. Ghezzi, M. Pradella, and G. Salvaneschi. An evaluation of the adaptation capabilities in programming languages. In *SEAMS*, pp. 50–59. ACM, 2011.

[17] E. Gjondrekaj, M. Loreti, R. Pugliese, and F. Tiezzi. Modeling adaptation with a data-driven coordination language. Technical report, Univ. Firenze, 2011. `http://rap.dsi.unifi.it/scel/pdf/MAWATBCL.pdf`.

[18] P. Greenwood and L. Blair. Using Dynamic Aspect-Oriented Programming to Implement an Autonomic System. In *DAW*, pp. 76–88, 2004.

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

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

[21] J. O. Kephart and D. M. Chess. The Vision of Autonomic Computing. *Computer*, 36:41–50, 2003.

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

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

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

[25] M. Parashar and S. Hariri. Autonomic computing: An overview. In *Unconventional Programming Paradigms*, pp. 247–259. Springer, 2005.

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

[27] 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*, *LNCS* 6721, pp. 202–218. Springer, 2011.