Modelling Constrained Dynamic Software Architecture with Attributed Graph Rewriting Systems

Cédric Eichler, Thierry Monteil, Patricia Stolf

Modelling Constrained Dynamic Software Architecture with Attributed Graph Rewriting Systems

Cédric Eichler[123], Thierry Monteil[23], and Patricia Stolf[13]

[1] IRIT; 118 Route de Narbonne, F-31062 Toulouse, France
[2] CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
[3] Univ de Toulouse, UPS, INSA, F-31400, UTM, F-31100 Toulouse, France

Abstract. Dynamic software architectures are studied for handling adaptation in distributed systems, coping with new requirements, new environments, and failures. Graph rewriting systems have shown their appropriateness to model such architectures, particularly while considering the consistency of theirs reconfigurations. They provide generic formal means to specify structural properties, but imply a poor description of specific issues like behavioural properties. This paper lifts this limitation by proposing a formal approach for integrating the consideration of constraints, non-trivial attributes, and their propagation within the framework of graph rewriting systems.

Keywords: Dynamic software architectures, Graph rewriting systems, Constrained and attributed rewriting systems

1 Introduction

Dynamic software architectures are studied for handling adaptation in autonomic distributed systems, coping with new requirements, new environments, and failures. The description of evolving architectures cannot be limited to the specification of a unique static topology but must cover the scope of all the correct configurations. This scope characterize an architectural style, qualifying what is correct and what is not. Naturally, once this distinction made, the question of the specification of the modifications themselves arises, alongside with theirs properties with regard to consistency preservation. These concerns motivate the need for suitable description languages and formalisms avoiding ambiguities for correct architectural design, management and analysis.

Several approaches have been proposed in the past adopting a language-based view of this problem. Many architecture description languages (ADLs) were introduced providing rigorous syntax and semantic to define architectural entities and relations [26, 22, 2, 13, 23]. Such languages are adequate to describe the structural and behavioural properties and constraints of a system. However, ADLs suffer from several insufficiencies for modelling and analysing dynamic software architecture [16]. The majority of ADLs are concentrated on the structural description of architecture whereas their dynamic aspects are not well supported. Darwin [23] only allows component replication whereas ACME [13] only allows optional components and connections. Dynamic-Wright [1] describes dynamic architecture in Wright [2], limiting itself to predefined dynamics, meaning that the system should have a finite number of configurations and reconfiguration policies known in advance.

Model-based approaches, proposing general-purpose modelling languages, allows to handle dynamism and particularly the definition of reconfiguration rules managing the evolution of an application in run-time. They provide very intuitive and visual formal or semi-formal description of structural properties [4]. For example, designing and describing software models using UML [25] is a common practice in the software industry, providing a standardized definition of system structure and terminology, as well as facilitating a more consistent and broader understanding of the architecture [31]. Nevertheless the generic fitness of model-based approaches implies poor means of describing specific issues like behavioural properties. Therefore they are often coupled with description using ADLs [29, 24, 5], mapping the concepts of ADLs into the visual notation of UML, or other formalisms [21, 17]. In spite of its wide acceptance, UML-based descriptions appears to lack expressiveness and formal tools for guaranteeing consistency. In fact, consistency checking in run-time may lead to combinatorial explosions. To tackle efficiently consistency preservation in the scope of dynamic reconfiguration, correct by design formal approaches have emerged. Based on formal proofs and reasoning in design-time, they guarantee the correctness of a dynamic application, requiring little or no verifications in run-time.

Graph-based methods for software modelling are appropriate for conceiving correct by design frameworks, as theoretical work on this field provides formal means to specify and check structural constraints and properties [30, 11]. Within this kind of approaches, some methods are restricted to the use of type graphs alone [33] and suffer from the same lack of expressiveness as UML-based methods. Other works [14, 20] are based on graph grammar, or graph rewriting system, techniques. Graph grammars are appropriate for formal modelling dynamic structures and software architectures, and are used to specify architectural style where a graph represents a configuration. Graph rewriting rules have two distinct values. They intervene in both the characterisation of an architectural style as part of a rewriting system and in the specification of consistency preserving reconfiguration rules [15]. The very first thing to consider is the definition of attributes for which two approaches prevail. The most simple one is to assign to each elements, vertices and edges, of a graph a list of couples representing constant or variable attributes along with theirs domains of definition [7]. The main drawback is that it does not allow to conduct any operation on said attributes or to fully propagate them. The second one is to define domains of definitions and operators in the form of a many sorted algebraic signature [12] SIG and to integrate attributes as vertices of the graph, elements of a SIG-algebra [3, 10]. A direct implication is a natural manipulation of attributes using operators and their addition or deletion as regular vertices of the graph. This modularity does not come without drawbacks. Graph rewriting rules relies on finding graph homomorphisms, a NP-complete problem. As a consequence, increasing the size of the input graphs seems inefficient, as well as multiplying the application of graph rewriting rules to modify attributes whenever a domino effect is implied.

The motivation of this paper comes from two simple statements. First, classical grammar theory allows complex propagation of attributes and the specification of elaborated semantic predicates on production rules. Propagation of attributes in graph grammars is very limited and often restricted to inheritance [7], restraining the means to model the impact of an event, such as a reconfiguration, on the attributes of an application. Semantic predicate are mostly present in the form of implicit system of simple equalities between attributes [7, 3, 10]. Secondly, these approaches focused on attributes and structural transformation, but disregard the appropriateness of a configuration. Yet constraints linked to such consideration are closely related to the architecture and its components. As a consequence, we wish, while constructing, deploying, or reconfiguring a configuration, to construct a set of constraints easy to evaluate. These lasts model some basic, more or less important, requirements of the application. As a result, their violation could be detected and automatically handled by a manager without requiring complex decision and without analysing the actual performances of the whole application.

A motivating example further enlightening the problems tackled by this article is presented in the next section. Section 3 introduces our formal model overcoming said obstacles. The characterisation of the motivating example is presented in Section 4, before discussing optimisation and evaluation issues. Finally, Section 5 is dedicated to conclusion and perspectives.

2 Problem Statement throughout a Motivating Example.

2.1 Motivating Example

In order to illustrate the issues addressed in this article, we introduce DIET[4] [6], a hierarchical load balancer for dispatching computational jobs over a distributed infrastructure, like a grid or a cloud. Its architecture consists of a set of agents: some Master Agents (MA) manage pools of computational SErver Deamons (SED) through none, one or several layers of Layer Agents (LA). These servers can achieve specialized computational services. Communications between agents are driven by the omniORB system (OMNI). MAs listen to client requests and dispatch them through the architecture to the best SED that can carry out

[4] Distributed Interactive Engineering Toolbox

the required service.
We stated previously that UML-based modelling suffers from a lake of expressiveness. This application has been described using class diagrams [32], but the fact that a LA can manage another LA could not be taken into consideration. It will be shown later that graph grammars do not exhibit such limitations.

A simplified architecture with a single MA and a single OMNI will be considered here. The main characteristics of the application are as follow :

1. While being deployed, each component record itself to the OMNI.
2. Each LA and each SED has a hierarchical superior.
3. The MA and each LA manage at least one LA or one SED and at maximum ten of them - these conditions could be trivially extended to any number of minimum and maximum managed entities, as seen later in this paper.

All instances of an architectural style are NOT created equals. At a given time, even though a configuration meet all the requirements of the application, another configuration may meet them in a "better way". In particular, we considered the following criteria : the energy consumption, the robustness, the fault-tolerance w.r.t. the breakdown of machine or a software component, and the quality of service. A very restricted view will be adopted here, with simple - simplistic, even - models as they still put the addressed problem under the spotlight.
Assume that the energy consumption depends only on the number of used machines and of the software components deployed on them.
Robustness is a little bit trickier. The first intuition is to exploit redundancy by deploying SEDs not having a disjoint set of carried out services. However, if every SED offering a particular service are deployed on the same machine, the application is still vulnerable to a machine breakdown. In a similar fashion, SEDs offering very close sets of services should not be managed by the same LA or LAs deployed on the same machine.
A single aspect of quality of service is studied : the balance of the managed entities over the set of LAs sharing the same depth, i.e. the number of intermediate LAs between a LA and the MA. Let LA(d) be the set of LA of depth d, and M(c) be the number of entities managed by the component c. An entity can be deployed and directly managed by a LA $\in$ LA(d) if it does not make the standard deviation of $\bigcup_{la \in LA(d)}$ M(la) become greater than an arbitrary value.
An interesting point here is that robustness and energy consumption are concurrent, in the sense that deploying more software components or using more machines will, while ameliorating the first, impact badly on the second.

To value these three conditions, it is crucial to keep track of some attributes of the software components :

1. the machine on which each entity is deployed,
2. the number of entities managed by each LA and the MA,
3. the depth of each LA,

4. the set of services carried out by each SED or by at least a SED managed directly or indirectly by each LA.

By misuse of language the set of services carried out by at least a SED managed directly or indirectly by an LA will sometimes be referred to as the set of services carried out by the LA.

2.2 Problem Statement

How is modelling such an architecture an issue? This seemingly simple problem still raises some interesting points underlining limitations of classical attributed graph grammars.

– *Interdependency of attributes*; the attributes of an entity depends on attributes of other entities. In classical attributed grammars, attributes are classified according to the fact that those other entities are parents or siblings of the first one , it is then said to be inherited, or not, in the case of a synthesized attribute. Similarly, those dependencies have to be handled in a very different way whether they rely on existing attributes or attributes of entities that has yet to be deployed. The first scenario, including the evaluation of the depth of an LA, which depends on the depth of the entity managing it, have been addressed with attributes inheritance in graph grammars. On the other hand, evaluating the set of services offered by at least a SED managed directly or indirectly by an LA can not be treated in the same fashion.
– *Modification of an existing attribute*; the option of considering attributes as nodes has been discarded due to its high implementation complexity. Consequently, graph rewriting rules cannot be used to modify attributes, and their modification are therefore not possible within the framework of classical graph grammars.
– *Conditional deployment*; whenever trying to deploy a LA or a SED, we should verify that it does not violate the maximal cardinality condition and the balancing condition. The first one could be handled in a structural, syntactic way using negative application conditions, implying other search for homomorphisms and thus a high computational complexity. The balancing condition, however, is too complex to be reasonably managed using pattern matching.
– *Evaluating a configuration*; we have seen that the constraints reflecting the appropriateness of a configuration are intimately related to attributes of the graph and do not weight on the same levels or components. Particularly, components of the same nature have close requirements that arise or evolve as components are deployed or as the context evolve, so that integrating such constraints in the model as any attributes is relevant. In addition, the entity managing the evolution of the architecture could then react quickly to their violation, without requiring much reasoning.

These four points put under the spotlight the limits of the formalism presented in [28] and the need for its expansion described in this paper.

3 Introducing Constraints and Mutators within Graph Rewriting Systems

3.1 Attributes, Constraints and Attributes Rewriting

Before introducing graph rewriting systems and how they may be used to model a dynamic software architecture, a certain number of concepts have to be defined. First of all, let us consider what an attribute shall be. We adopt an hybrid formalism conserving the simplicity and the computational efficiency of "listing" attributes as labels while granting the possibility of applying any operator. Informally, an attribute is represented as a constant, a variable or the result of any function on other attributes, as long as the signature of the function is respected, coupled with its domain of definition. The expression of the function may includes quantifiers. We assume the canonic notation where Y^X is the set of function from X to Y.

Definition 1. ***(Attribute)*** *An attribute is a couple $Att = (Att_A, Att_D)$ where*

- *Att_A is called value and is either*
  - *a variable,*
  - *a constant or*
  - *an expression, a regular combination of attributes : $\forall n \in \mathbb{N}, \forall$ sequence of sets $(D_u)_{u\in[|1,n|]}$, $\forall\ f \in D^{D_1\times D_2\times\cdots\times D_n}$, a couple $Att = (f(A_1,..,A_n), Att_D)$ is a regular combination of attributes of $(A_i,\ D_{A_i})_{i\in[|1,n|]}$ if and only if $Att_D \subseteq D$ and $\forall i \in [|1,n|]$, $(A_i,\ D_{A_i})$ is a (defined) attribute and $D_{A_i} \subseteq D_i$.*
- *and Att_D its domain of definition.*

An attributed structure is a couple composed of the structure and a set of indexed attributes or sequence of attributes. The first member of an attribute will be noted within quotation marks if and only if it is constant. The fact that the first member of an attribute is constant does not implies that it can not change any further, but rather that at a given time the attribute is equals to a member of its domain of definition.

Remark 1. In the approach presented here the attributes can be a regular combination of any other attributes, not necessary the attributes of the same structure. For example, considering an attributed structure (A, ATT_A) of attributed structures (B_i, ATT_{B_i}) of attributed elements (C^i_j, $\mathrm{ATT}_{C^i_j}$). Let $\mathrm{ATT} = ATT_A \cup \bigcup_i ATT_{B_i} \cup \bigcup_{(i,j)} ATT_{C^i_j}$. $\forall$ Att $\in$ ATT, $\forall N \in \mathbb{N}$, $\forall$ $(\mathrm{Att}_k)_{k\in[|1,N|]} \in ATT^N$, Att can be a regular combination of $(\mathrm{Att}_k)_{k\in[|1,N|]}$. From now on, we adopt the definition

Remark 2. Note that this notation is related to the approach considering many-sorted signatures and algebras as elements are attributed over an implicit SIG-algebra. Informally, the sorts of SIG are every defined domains of definition and its operations are every functions from any combination of sorts to any sort.
For any attributed structure (*str*, ATT_{str}), let ATT be the set of attributes resulting of the union of any attribute defined in the framework of *str*, which is not empty as at least $\mathrm{ATT}_{str} \subseteq \mathrm{ATT}$. Let S = { s |∃ Att, (Att, s) ∈ ATT} and OP = { f | $\exists N \in \mathbb{N}, \exists (Att_i)_{i \in [|0,N|]}$, f ∈ $Att_0^{Att_1 \times Att_2 \times \cdots \times Att_N}$ }. *str* is attributed over a (S,OP)-algebra.

Attributes are entirely aimed at providing informations on an algebraic structure. It is very natural to desire to exploit these bits of information by expressing any property over a structure or its attributes. Those having been defined in a very generic way, constraints can be seen as a specific kind of attributes taking values in a ternary predicate logic system.

Definition 2. ***(Constraint)*** *A constraint Cons = (C, D) is an attribute (C, D) and D = { "true", "false", "unknown"}.*

To simplify the notation, considering that constraints share the same domain of definition, a constraint Cons = (C, D) may be referred to as C, and the domain of definition be implicit. In the following we adopt the principles of Kleene's strong logic [18] [19], in particular its basic logic operations ($\vee, \wedge, \neg, \implies$) and the fact that the only truth value is "true". The uniqueness of this truth value means that evaluations are pessimistic, i.e. "unknown" is supposed to be false.

Remark 3. A constraint, as a regular combination of attributes, can be seen as a classical expression of a predicate ternary logic. Considering a ternary logic rather than a binary one implies that unlike attributes, constraints can be evaluated, i.e. associated with a constant, at any time. The idea here is to associate any minimal logic expression that can not be evaluated, due for example to an attribute implied in its expression being un-evaluable or variable, with "unknown".

Example 1. Let S be any mathematical structure along with the set of attributes ATT = {Att_1, Att_2, Att_3} = {(A_1, Mach), (A_2, Mach), (A_3, {"true", "false", "unknown"})} where A_1 = m, A_2 = "$\bar{m}$", and A_3 = "true" and the set of constraints CONS = {C_1, C_2 } = { ($\mathrm{A}_1 \neq \mathrm{A}_2$), ($\mathrm{C}_1 \vee A_3$) }. Even though Att_1 can not be evaluated to a constant, C_1 can be evaluated to "unknown". C_2 is then equals to "unknown" ∨ "true", which is "true".

In order to lighten the notation, an attributed system with constraints, i.e. a triple composed by the system, a set of indexed attributes or sequence of attributes, and a set of indexed constraints or sequence of constraints, is noted an AC-system. Whenever defining an AC-system with AC-elements, rather than separating each sets of attributes (resp. constraints), a single family of sequence of attributes indexed by the sets of attributed elements is considered.

One of the issues evoked in section 2 is the fact that attributes are prone to evolve. A reconfiguration may thus impact the attribute of the system. Classical string rewriting theory [27] tackles this issue by using mutators. A similar approach is adopted here.

Definition 3. ***(A mutator on an AC-system)*** *A mutator on an AC-system is an arbitrary algorithm updating the value(s) of none, one or some attributes of the AC-system.*

According to this definition, the scope of mutators remains limited to the modification of values. They can not be used neither to add or suppress an attribute nor to modify the domain of definition of an attribute.

3.2 Attributed Constrained Graph Modelling a Configuration

Tools to manipulate and analyse attributes having been introduced, the structures of interest can be presented. An AC-graph, modelling a software snapshot or configuration at a given time, consists in an AC-couple of two AC-sets of vertices and edges where an edge is a couple of vertices (source, destination). Following the commonly used conventions for standard graphical descriptions, one considers that vertices represent services or architectural components and edges correspond to their related interdependencies. Note that vertices, edges and the graph itself are AC-structures. According to remark 1, elements are attributed over the same algebra, i.e. an attribute or a constraint of the graph, a vertex or an edge may consist in any regular combination of attributes of the graph, any vertex and any edge. For any set S, the cardinality of S is represented as |S|.

Definition 4. ***(AC-graph)*** *An AC-graph is defined by the system G = (V, E, ATT, CONS) where*

- *V and $E \subseteq V^2$ correspond respectively to the set of vertices and edges of the graph,*
- *ATT (resp. CONS) is a family of sets ATT_{el} (resp. $CONS_{el}$), indexed by a subset of $V \cup E \cup \{G\}$. ATT_{el} is a set of attributes (resp. constraints) of arbitrary length and containing the sequence of attributes $(ATT^i_{el} = (A^i_{el}, D^i_{el}))_{i\in[|1,|ATT_{el}||}$ (resp. $(CONS^i_{el} = (C^i_{el}, D^i_{el}))_{i\in[|1,|CONS_{el}||}$) of the element el.*

ATT and CONS are indexed by a subset of $V \cup E \cup \{G\}$, so as not to impose an empty set of attributes or constraint if an element is wished not to be attributed or constrained. Consequently, $0 \leq |ATT| \leq |V| + |E| + 1$ and $0 \leq |CONS| \leq |V| + |E| + 1$. Now that AC-graphs are defined, it is possible to represent a configuration of DIET as presented in section 2.

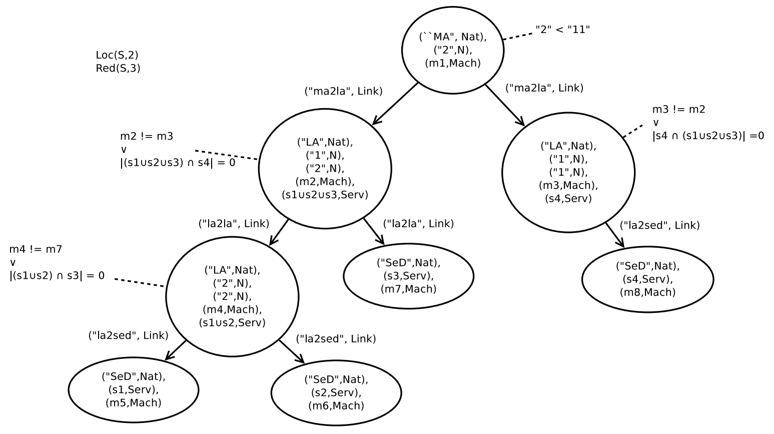
Loc(S,2)
Red(S,3)
(``MA", Nat), ("2",N), (m1,Mach)
"2" < "11"
("ma2la", Link)
("ma2la", Link)
m2 != m3 v |(s1∪s2∪s3) ∩ s4| = 0
("LA",Nat), ("1",N), ("2",N), (m2,Mach), (s1∪s2∪s3,Serv)
m3 != m2 v |s4 ∩ (s1∪s2∪s3)| =0
("LA",Nat), ("1",N), ("1",N), (m3,Mach), (s4,Serv)
("la2la", Link)
("la2la", Link)
("la2sed", Link)
m4 != m7 v |(s1∪s2) ∩ s3| = 0
("LA",Nat), ("2",N), ("2",N), (m4,Mach), (s1∪s2,Serv)
("SeD",Nat), (s3,Serv), (m7,Mach)
("SeD",Nat), (s4,Serv), (m8,Mach)
("la2sed", Link)
("la2sed", Link)
("SeD",Nat), (s1,Serv), (m5,Mach)
("SeD",Nat), (s2,Serv), (m6,Mach)

Fig. 1. An AC-graph modelling a configuration of DIET

Example 2. Figure 1 represents an AC-graph modelling some configuration of DIET, where the naming server has been omitted for clarity concerns. The graph is formally defined in Appendix A.

The notations used in Fig. 1 are as follow; Mach is the set of available machines, on which a software component may be deployed, Nat the set of possible nature of a software component, {"OMNI", "MA", "LA", "SED"} in the case of a DIET architecture, Link the set of possible relationships {"ma2la", "ma2sed", "la2sed", "registered"}, and Serv the power set of S, the set of services that could be carried out by a SED. At this time, the software architecture is composed by eight components symbolized by eight vertices and theirs corresponding relations modelled by some edges both with theirs attributes. A notable fact is that components of the same nature have the same number of attributes, theirs attributes being the one identified in Sect. 2. This stem from the definition of the rewriting system that will be presented later in this paper. Attributes have similar meanings, be it the number of managed entities, the depth of a LA...
In addition, some components as well as the graph itself are constrained to reflect the concerns stated in the same section. Graphically, a dot line between a constraint and a vertex illustrates the target of said constraint. The MA should manage strictly less than eleven entities, underlining a fundamental property of the architectural style. Even though the maximum number of entities managed by the LAs could be expressed in the same way, an other technique will be used and presented later. To cope with robustness, it is specified that a LA should not be deployed on the same machine as a component managed by the same entity as itself or that they should have disjoint set of carried services. The graph is constraints by two clauses Loc and Red, described in Appendix A, taking into account the needs for redundancy and multiple localisations over the offered services.

This is how a DIET configuration should look like. From now on, our aim is to introduce the notions allowing to characterize the corresponding architectural style, ensuring in particular that attributes are correctly updated and that components have the required constraints.

3.3 Graph Rewriting Rules and Systems

As stated in the introduction, an architectural style can be formalised using a graph rewriting system. The production rules of such systems require to identify sub-structures by the means of homomorphisms. An unattributed graph homomorphism between two graphs is defined as an homomorphism from the set of vertices of the first one to the set of vertices of the second graph so that if there is an edge between two vertices of the first one there is an edge between their image in the second one. Based on [28], we define an attributed graph homomorphism as follow.

Definition 5. ***(AC-graph homomorphism)*** *Two AC-graphs G = (V, E, ATT, CONS) and G' = (V', E', ATT', CONS') are homomorph -noted $G \rightarrow G'$- if and only if there is a graph-homomorphism h from (V, E) to (V', E') such as*

1. *$\forall\ v \in\ V$ (resp. $\forall\ e = (\bar{v}, \tilde{v}) \in E$), $|ATT_v| = |ATT_{h(v)}|$ (resp. $|ATT_e| = |ATT_{(h(\bar{v}),h(\tilde{v})}|$),*
2. *$\forall\ v \in V$ (resp. $\forall\ e = (\bar{v}, \tilde{v}) \in E$), $\forall\ i \in [\![1, |ATT_v|]\!]$, $D^i_v = D^i_{h(v)}$,*
3. *The system of equations $S = \{\ A = A' \mid (\exists\ v \in V, \exists\ i \in [\![1, |ATT_v|]\!], A = A^i_v \wedge A' = A^i_{h(v)}) \vee (\exists\ e = (\bar{v}, \tilde{v}) \in E, \exists\ i \in [\![1, |ATT_e|]\!], A = A^i_e \wedge A' = A^i_{(h(\bar{v},h(\tilde{v}))}) \ \}$ has at least one solution.*

Remark 4.
- Constraints do not impact on the definition of an homomorphism. It will be shown that they intervene in the rewriting process in a different way. Similarly, attributes on vertices and edges are the only one that are considered whereas attributes on the graph itself are not.
- We underline the fact that the existence of an AC-homomorphism is conditioned by the resolvability of a system of equations on attributes. As stated in the introduction, in [3, 21, 14, 20] and most of the research on attributed graphs, the existence of a morphism is also conditioned by equalities between attributes, potentially through morphism between attributes spaces. However, this is often the only clause relying on attributes that impact the applicability of a graph rewriting rule.
- AC-graph isomorphism and AC-vertex morphism can be trivially deduced from the definition of AC-graph homomorphism.

Solving the system of equation S result in identifying the value of some attributes with some constants in theirs domains of definition and/or with some other attributes. Integrating the affectation obtained by solving this system refers to the update of the value of the attribute to reflect these identifications. Intuitively, such an integration is nothing more than a dimension reduction of the space formed by the potential values of the attributes. For more information see [28].

There exists a vast number of approaches handling graph rewriting based on attributed graphs [30, 9]. Their applicability depends on various factors, suspension condition, negative application condition(s), always including the existence of an homomorphism between an element of the graph rewriting rule and the graph to rewrite. Inspired by string grammar theory [27], we include in these factors the satisfaction of a, potentially empty, set of constraints on attributes, namely the set of constraints of the AC-rewriting rule. Consequently, the set of constraints on a graph rewriting rule can be seen as a set of semantic predicates.

Applying a rewriting rule on a graph consists in suppressing a part of the graph and extending it by adding some vertices and edges. In addition to classical modifications induced by the application of a rule, a set of actions is performed at the end of said application.

Virtually, any attributed graph rewriting formalism could be extended to include semantic predicates, constraints and mutators. In order to fix the idea, we chose the classical double push out formalism defined in [30] alongside with the attribute management considered previously and used in [28].

Definition 6. ***(AC-graph AC-rewriting rule)*** *An AC-graph AC-rewriting rule is a 5-tuple (L, K, R, ATT, CONS, ACT) where*

- $ATT = ATT_L \cup\ ATT_R$ *is a set of attributes,* ATT_{rule} *being the set of attributes of the graph rewriting rule itself,*
- $CONS = CONS_{rule} \cup\ CONS_{R\setminus K}$ *is a set of constraints,* $CONS_{rule}$ *being the set of constraints of the graph rewriting rule itself and* $CONS_{R\setminus K}$ *the set of constraints of* $R\setminus K$*,*
- $(L = (V_L,\ E_L),\ ATT_L,\ \emptyset)$ *and* $(R = (V_R,\ E_R),\ ATT_R,\ CONS_{R\setminus K})$ *are AC-graphs,*
- $K = (V_K,\ E_K)$ *is a sub-graph of both L and R,*
- *ACT is a set of actions.*

$L\setminus K$ *is called the Del zone and* $R\setminus K$ *is called the Add zone, while K is called the Inv zone.*

Furthermore, graph rewriting rules are illustrated here using the delta representation, where only one graph is considered. This graph is visually partitioned into three zones, from left to right the Del, Inv and Add zones. This graphical representation is illustrated in Fig. 2.

A rule is applicable on a AC-graph G if :

1. there is an AC-homomorphism h : (L, ATT_L, CONS_L) $\rightarrow$ G, implying in particular that the system of equations S = $\{$ A = A' $|$ ($\exists$ v $\in$ V_L, $\exists$ i $\in$ [|1, $|\mathrm{ATT}_v|$|], A = A_v^i $\wedge$ A' = $\mathrm{A}^i_{h(v)}$) $\vee$ ($\exists$ e = $(\bar{v}, \tilde{v})$ $\in$ E, $\exists$ i $\in$ [|1, $|\mathrm{ATT}_e|$|], A = A_e^i $\wedge$ A' = $\mathrm{A}^i_{(h(\bar{v},h(\tilde{v}))}$) $\}$ has at least a solution,

2. the application of the rule would not lead to the apparition of any dangling edge,
3. each Cons $\in$ CONS$_{rule}$ is evaluated to “true” by integrating the affectations obtained by solving S and by evaluating each elementary logic expression containing variable attributes to “unknown” as stated in remark 3.

Its application consists in :

1. erasing h(L\K),
2. integrating the affectations obtained by solving S to the remaining graph,
3. adding an isomorph copy of R\K integrating the affectations obtained by solving S,
4. performing each action Act $\in$ ACT.

Inspired from Chomsky’s generative grammars [8], graph grammars are defined as a classical rewriting system and formally characterize an architectural style.

Definition 7. ***(Graph Grammar)****A graph grammar is defined by the 4-tuple* (AX, NT, T, P) *where*

– AX *is the axiom, an AC-graph with a single vertex* AX
– NT *is the sets of non-terminal archetypes of AC-vertices,*
– T *is the set of terminal archetypes of AC-vertices,*
– P *is the set of AC-graph AC-rewriting rules, or production rules, belonging to the graph grammar.*

Each vertex occurring in a graph rewriting rule in P *or in a graph obtained by applying a sequence of productions* $\in P$ *to the axiom is then isomorph to at least one arch-vertex in* NT *or* T*.*

Remark 5. Semantic predicates allow to easily guarantee the finition of such rewriting systems, by imposing, for example, a limited number of deployed components.

Definition 8. ***(Instance belonging to the graph grammar)*** *An instance belonging to the graph grammar* (AX, NT, T, P) *is a graph whose vertices and edges have only constant attributes and obtained by applying a sequence of productions in* P *to* AX*. If an instance does not contain any vertex isomorph to an arch-vertex from* NT *it is said to be consistent.*

Graph rewriting systems are illustrated in the next section, where they are used to characterize the example presented in Sect. 2.

4 Characterisation of the Motivating Example.

By characterising the motivating example using the formalism presented in the previous section, it appears undoubtedly that it addresses the problems identified in Sect. 2.

4.1 Components arch-types : Terms of the Grammar

Firstly, let us consider what the axiom shall be. Considering the definition of graph rewriting rules and systems, instances of the such systems are graphs that inherit the attributes and constraints of the axiomatic graph. We wish to expressed the redundancy and localisation constraints on instances of the architectural style, as described in appendix A, as well as the maximum number of entities that a MA or a LA may manage and the arbitrary value intervening in the balancing condition. Therefore, let AX_{DIET} be (v_{AX}, ATT_{AX} = ((maxSonsLA, $\mathbb{N}$), (maxSonsMA, $\mathbb{N}$), (maxσ, $\mathbb{R}^+$)), $CONS_{AX}$ = (L(S,2),R(S,3))), where here maxSonsMA = maxSonsLA = 10. Throughout this section the graph, on which production rules will be attempted to be applied to, is noted G = (V, E, ATT, CONS). Note that as stated in this paragraph, if G is an instance of the architectural style defined here, ATT = ATT_{AX} and $CONS_G$ = $CONS_{AX}$.

Secondly, terminal terms of the graph rewriting system characterizing DIET have to be defined. As they are AC-vertices, defining their attributes and constraints is sufficient. Remember that these term are arch-vertices and defined a pattern of attributes and constraints for the vertices of each instance. The first ones have already been listed in Sect. 2, while constraints stem from considerations exposed in the same section. The naming system itself is not constrained and is quite simple, let T_{Omni} be (v_{Omni}, ATT_{Omni} = ((“Omni”, Nat), (m, Mach)), $\emptyset$). Similarly, let T_{SeD} = (v_{SeD}, ATT_{SeD} = ((“SeD”, Nat), (s, Serv), (m, Mach)), $\emptyset$). The MA shall not manage more than 10 entities. Accordingly, let T_{MA} be (v_{MA}, ATT_{MA} = ((“MA”, Nat), (Nsons, $\mathbb{N}$), (m, Mach)), $CONS_{MA}$ = (($A_2 < ATT^2_{AX}$))). Finally, we stated before that LAs of the same layer should not be deployed on the same machine or have disjoint set of carried out services. Hence, let T_{LA} be (v_{LA}, ATT_{LA} = ((“LA”, Nat), (depth, $\mathbb{N}$), (Nsons, $\mathbb{N}$), (m, Mach), (s, Serv)), $CONS_{LA}$ = c(v_{LA})), where $c(v) = (c(v)_i)_{i \in [|1,|LA((ATT_G)^2_v)|-1|]}$ and $\forall \tilde{v} \in LA((ATT_G)^2_v)\backslash\{v\}$, $!\exists$ i $\in [|1, |LA((ATT_G)^2_v)| - 1|]$, $c(v)_i = ((ATT_G)^4_v \neq (ATT_G)^4_{\tilde{v}}) \vee ((ATT_G)^5_v \cap (ATT_G)^5_{\tilde{v}} = \emptyset)$.

4.2 Productions of the Rewriting System

Production rules of the graph grammar formalize the construction of instances by defining when and how an entity may be deployed and the consequences of such a deployment. The first rule to define is the initialisation consuming the axiomatic vertex (Del). The naming service and the MA are deployed, as well as a non-terminal vertex granting that the MA manages at least an entity (Add). This vertex will be later on instantiated into a LA or a SeD. Finally, the MA register to the naming service through the action register. Let p_1 = (L_{p_1}, K_{p_1}, R_{p_1}, $\emptyset$, $\mu_{registering}$(pv2)), where L_{p_1}, K_{p_1}, R_{p_1} and pv2 are defined in Fig. 2 and $\mu_{registering}$(v) is the action of registering the object represented by the vertex v to the naming service.

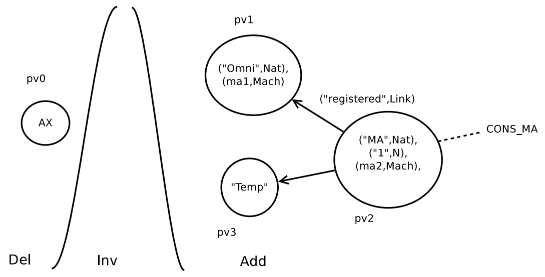

Fig. 2. Initialisation

Productions rules p_2 and p_3 model the addition of a non-terminal vertex, that will later on be instantiated into a LA or a SeD, and manage respectively by the MA or a LA. To deploy a new entity under a LA, the balancing condition and the maximal number of managed entities should be respected. The application of one of these productions leads to the incrementation of the number of sons of the entity on which the terminal vertex is added. Let $p_2 = (L_{p_2}, K_{p_2}, R_{p_2}, \emptyset, \mu_{inc}(\text{pv2MA},2))$ and $p_3 = (L_{p_3}, K_{p_3}, R_{p_3}, (\text{balancing(pv2LA)}, \text{Nsons} < \text{ATT}^1_{AX}), \mu_{inc}(\text{pv2LA},3))$ where L_{p_2}, K_{p_2}, R_{p_2}, pv2LA, L_{p_3}, K_{p_3}, R_{p_3}, and pv2MA are defined in Fig. 3, while μ_{inc}(v, i) is described below and balancing(v) = $\sigma(((\text{ATT}_G)^3_{la})_{la \in LA(d) \setminus \{v\}}, (\text{ATT}_G)^3_v+1) < \text{ATT}^3_{AX}$, where σ(s) is the standard deviation of the sequence s.

```
$\mu_{inc}(v,\ i)$
    $\text{Att}^i_v \leftarrow \text{Att}^i_v + 1$
```

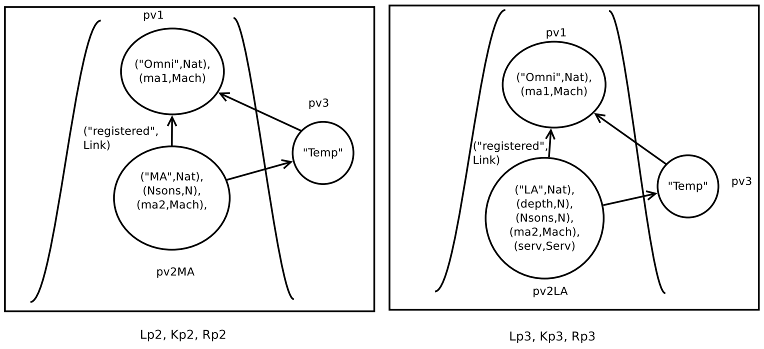

Fig. 3. Addition of a non-terminal term

The instantiation of a temporary vertex into a SeD is described by p_4 and p_5 if it is managed by respectively the MA or a LA. After deploying the SeD, it has to register to the naming service and, if it is managed by a LA, an update of carrying out services must be conducted. Let p$_4$ = (L$_{p_4}$, K$_{p_4}$, R$_{p_4}$, $\emptyset$, $\mu_{registering}$(pv4)) and p$_5$ = (L$_{p_5}$, K$_{p_5}$, R$_{p_5}$, $\emptyset$, ($\mu_{registering}$(pv4), $\mu_{updateServ}$(pv2,pv4)), where L$_{p_4}$, K$_{p_4}$, R$_{p_4}$, L$_{p_5}$, K$_{p_5}$, R$_{p_5}$, pv2, and pv4 are defined in Fig. 4, and $\mu_{updateServ}$(v, $\tilde{v}$) is as follows.

```
$\mu_{updateServ}(v, \tilde{v})$
    oldServ $\leftarrow (\mathrm{Att}_G)^5_v$
    ind $\leftarrow 5$
    if $(\mathrm{A}_G)^1_{\tilde{v}}$ = "SeD"
        ind $\leftarrow 2$
    $(\mathrm{Att}_G)^5_v \leftarrow$ oldServ $\cup (\mathrm{Att}_G)^{ind}_{\tilde{v}}$
    if $(\mathrm{Att}_G)^5_v \neq$ oldServ
        $\bar{v} \leftarrow v' \in \mathrm{V}_G, (v', \bar{v}) \in \mathrm{V}_E$
        $\mu_{updateServ}(\bar{v}, v)$
```

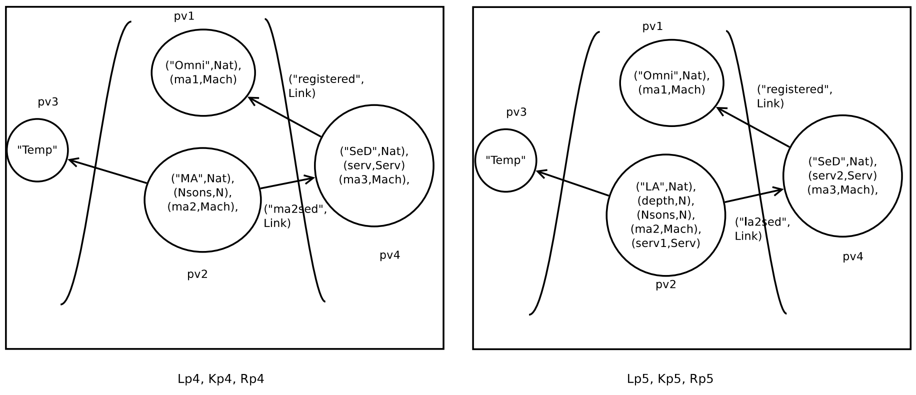

Fig. 4. Instantiation of a non-terminal term into a SeD

The two last productions of the grammar, p$_6$ and p$_7$, describe the instantiation of non-terminal term into a LA managed respectively by the MA or a LA. Let p$_6$ = (L$_{p_6}$, K$_{p_6}$, R$_{p_6}$, $\emptyset$, $\mu_{registering}$(pv4)) and p$_7$ = (L$_{p_7}$, K$_{p_7}$, R$_{p_7}$, $\emptyset$, $\mu_{registering}$(pv4)), where L$_{p_6}$, K$_{p_6}$, R$_{p_6}$, L$_{p_7}$, K$_{p_7}$, R$_{p_7}$, and pv4 are defined in Fig. 5.

4.3 The Constrained Attributed Graph Rewriting System Characterizing the Motivating Example

Considering the definition introduced in this section, GRS$_{DIET}$, the graph rewriting system formally characterizing the motivating example introduced in Sect. 2, is defined as GRS$_{DIET}$ =(AX$_DIET$, NT$_{DIET}$, T$_{DIET}$, P$_{DIET}$), where NT$_{DIET}$ = (v$_{temp}$, ATT$_{temp}$ = ("temp", {"temp"}), CONS$_{temp}$ = $\emptyset$),

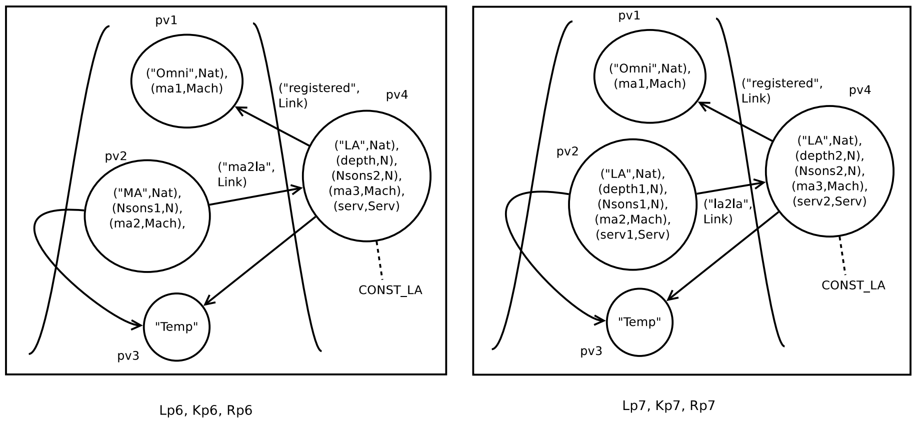

Fig. 5. Instantiation of a non-terminal term into a LA

$T_{DIET} = \{ T_{Omni}, T_{MA}, T_{LA}, T_{SeD}\}$, and
$P_{DIET} = \{ p_1, p_2, p_3, p_4, p_5, p_6, p_7 \}$.

Remark 6. Efficiently representing evolving attributes. An interesting feature of this formalism is that it offers several options for the characterization of problematic attributes or constraints. Let us consider the set of services that may be carried out by at least a SeD managed directly or indirectly by a LA, for example. Intuitively, this attribute depends on other ones and could be expressed as a regular combination of the graph attributes. However, this combination should have been evaluated whenever its value is required, for example to evaluate the constraints of a LA. To avoid frequent evaluations, the graph representing the configuration can be seen as an augmented data structure and the attribute be updated whenever it has to be using mutators. The choice between these two options rely on the complexity and frequency of updates and evaluations.

This model allows to express each limitation identified in Sect. 2, interdependency between attributes, modification of existing attributes and non-trivial requirements for conditional deployments. Furthermore, components of the architecture are constrained to reflect the appropriateness of a configuration. These constraints are inherited from arch-vertices as a component is deployed, and may evolve like any attribute to reflect context changes.

Evaluation of a configuration can be done in several ways. The most straightforward would be to assign a, potentially infinite, weight to the violation of constraints to define management policies through heuristics. However, constraints on an architecture and their satisfaction can be seen as a way of granting some desirable properties rather than looking for an optimum. Considering that finding a “good” configuration as a classical multi-criteria optimisation problem, it could be solved by restraining it to a single-criteria, here energy consumption, problem over the space of consistent configurations that does not violate any constraint.

5 Conclusion

This article lift limitations of formal approaches for the characterisation of dynamic software architectures using graph rewriting systems. Said limitations are put under the spotlight using an example of hierarchical dynamic software architecture that previous approaches have failed to characterize, and that later on serve as an illustration for our model. In particular, interdependency of attributes is handled using regular combination of attributes, and mutators are introduced to modify existing attributes. Consequently, non-trivial attributes related to structural and behavioural properties of the architecture may be handily represented, as shown in the example. Based on such attributes, dynamic constraints are included in the formal model, reflecting the appropriateness of a configuration, arising or evolving as components are deployed or as the context change. Moreover, further properties can be guaranteed as the formalism presented in this paper tackles non-trivial conditional deployments using semantic predicates.

Constraints and attributes facilitate the evaluation of a configuration, a basis for the definition and the enforcement of reconfiguration policies. We plan on developing the mechanisms evoked in this paper to achieve these aims, as well as extending existing tools to realise management using this formalism and validate its time efficiency with regard to other graph-based approaches.

Acknowledgement

The work presented in this paper has been funded by the ANR in the context of the project SOP, ANR-11-INFR-001.

A Formal definition of the graph presented in Fig 1

The graph in the Fig. 1 is defined as follow.

G = (V, E, ATT, CONS) where
V = {v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8},
E = { e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_4), e_4 = (v_2, v_7), e_5 = (v_3, v_8), e_6 = (v_4, v_5), e_7 = (v_4, v_6)},
ATT = {ATT_G, ATT_{v_1}, ATT_{v_2}, ATT_{v_3}, ATT_{v_4}, ATT_{v_5}, ATT_{v_6}, ATT_{v_7}, ATT_{v_8}, ATT_{e_1}, ATT_{e_2}, ATT_{e_3}, ATT_{e_4}, ATT_{e_5}, ATT_{e_6}, ATT_{e_7}, ATT_{e_8}},
The architecture is composed by some software component, the first one being v_1, a MA managing 2 entities and deployed on a machine noted m_1 ATT_{v_1} = {(“MA”, Nat), (“2”, $\mathbb{N}$), (m_1,Mach)}.

Three LAs are deployed, respectively of depth 1, 1 and 2, managing 2, 1 and 2 entities, placed on machine m_2, m_3 and m_4. The first one managed directly or indirectly SEDs providing the set of services $s_1 \cup s_2 \cup s_3$, the second one s_4 and the third one $s_1 \cup s_2$.
ATT_{v_2} = {("LA", Nat), ("1", $\mathbb{N}$), ("2", $\mathbb{N}$), (m_2, Mach), ($s_1 \cup s_2 \cup s_3$, Serv)},
ATT_{v_3} = {("LA", Nat), ("1", $\mathbb{N}$), ("1", $\mathbb{N}$), (m_3, Mach), (s_4, Serv)},
ATT_{v_4} = {("LA", Nat), ("2", $\mathbb{N}$), ("2", $\mathbb{N}$), (m_4, Mach), ($s_1 \cup s_2$, Serv)}.

Finally, four SEDs deployed on m_5, m_6, m_7 and m_8 carry out the services s_1, s_2, s_3 and s_4.
ATT_{v_5} = {("SED",Nat), (s_1, Serv), (m_5, Mach)},
ATT_{v_6} = {("SED",Nat), (s_2, Serv), (m_6, Mach)},
ATT_{v_7} = {("SED",Nat), (s_3, Serv), (m_7, Mach)},
ATT_{v_8} = {("SED",Nat), (s_4, Serv), (m_8, Mach)}.

The graph itself is constraints by L(S,2) and R(S,3), where R and L are as follows. $\forall S' \subseteq S, \forall x_s \in \mathcal{N}$, let the redundancy constraint R(S',x_s) be "There are at least x_s SeDs carrying each service s in S'".
Red(S',x_s) = $\forall s \in S', \exists (V_i)_{i \in [|1,x_s|]} \in V_A^{x_s}$, ($\forall (i,j) \in [|1,x_s|]^2$, $i \neq j \implies V_i \neq V_j) \wedge \forall k \in [|1,x_s|]$, $s \in ATT^2_{V_k}$.
$\forall s \in S, \forall x_s \in \mathcal{N}$, let the localisation constraint L(S',x_s) be "For each service in S', there are at least x_s different machines on which at least a SeD carrying out the service s is deployed".
Loc(S',x_s) = $\forall s \in S', \exists (V_i)_{i \in [|1,x_s|]} \in V_A^{x_s}$, ($\forall (i,j) \in [|1,x_s|]^2$, $i \neq j \implies (V_i \neq V_j \wedge ATT^3_{V_i} \neq ATT^3_{V_j}) \wedge \forall k \in [|1,x_s|]$, $s \in ATT^2_{V_k}$.

This means that each service should be carried out by at least 3 SEDs located on at least two different machine. The MA has a constraints on the number of entities it manages. The two LA of depth 1 are constrained not to be deployed on the same machine or to offer disjoint sets of services. The last LA has a similar constraint.
CONS = {$CONS_G$, $CONS_{v_1}$, $CONS_{v_2}$, $CONS_{v_3}$, $CONS_{v_4}$},
$CONS_G$ = {L(S,2), R(S,3)},
$CONS_{v_1}$ = {$ATT^2_{v_1} \leq 10$},
$CONS_{v_2}$ = {$ATT^4_{v_2} \neq ATT^4_{v_3} \vee (ATT^5_{v_2} \cap ATT^5_{v_3} = \emptyset)$ },
$CONS_{v_3}$ = {$ATT^4_{v_3} \neq ATT^4_{v_2} \vee (ATT^5_{v_3} \cap ATT^5_{v_2} = \emptyset)$ } and
$CONS_{v_4}$ = {$ATT^4_{v_4} \neq ATT^3_{v_7} \vee (ATT^5_{v_4} \cap ATT^2_{v_7} = \emptyset)$ }

References

1. Allen, R., Douence, R., Garlan, D.: Specifying and analyzing dynamic software architectures. In: Astesiano, E. (ed.) Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol. 1382, pp. 21–37. Springer Berlin Heidelberg (1998), http://dx.doi.org/10.1007/BFb0053581

2. Allen, R., Garlan, D.: A formal basis for architectural connection. ACM Trans. Softw. Eng. Methodol. 6, 213–249 (July 1997), http://doi.acm.org/10.1145/258077.258078
3. Berthold, M.R., Fischer, I., Koch, M.: Attributed graph transformation with partial attribution. Berkeley Initiative in Soft Computing , University of California at Berkeley http://europepmc.org/abstract/CIT/302744
4. Bradbury, J.S., Cordy, J.R., Dingel, J., Wermelinger, M.: A survey of self-management in dynamic software architecture specifications. In: Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems. pp. 28–33. WOSS '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1075405.1075411
5. Broto, L., Hagimont, D., Stolf, P., de Palma, N., Temate, S.: Autonomic management policy specification in tune. In: ACM Symposium on Applied Computing. pp. 1658–1663. Fortaleza, Ceara, Brazil (2008)
6. Caron, E., Desprez, F.: Diet: A scalable toolbox to build network enabled servers on the grid. International Journal of High Performance Computing Applications 20(3), 335–352 (2006)
7. Chassot, C., Guennoun, K., Drira, K., Armando, F., Exposito, E., Lozes, A.: Towards autonomous management of qos through model-driven adaptability in communication-centric systems. ITSSA 2(3), 255–264 (2006)
8. Chomsky, N.: Three models for the description of language. Information Theory, IEEE Transactions on 2(3), 113–124 (1956), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1056813
9. Ehrig, H.: Tutorial introduction to the algebraic approach of graph grammars. In: Ehrig, H., Nagl, M., Rozenberg, G., Rosenfeld, A. (eds.) Graph-Grammars and Their Application to Computer Science, Lecture Notes in Computer Science, vol. 291, pp. 1–14. Springer Berlin Heidelberg (1987), http://dx.doi.org/10.1007/3-540-18771-5_40
10. Ehrig, H., Ehrig, K., Prange, U., Taentzer, G.: Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundam. Inf. 74(1), 31–61 (Oct 2006), http://dl.acm.org/citation.cfm?id=1231199.1231202
11. Ehrig, H., Kreowski, H.J.: Graph Grammars and Their Application to Computer Science: 4th International Workshop, Bremen, Germany, March 5-9, 1990 Proceedings. Springer-Verlag New York, Inc., Secaucus, NJ, USA (1991)
12. Ehrig, H., Mahr, B.: Fundamentals of algebraic specification. EATCS monographs on theoretical computer science, Springer-Verlag, Berlin, New York (1985), http://opac.inria.fr/record=b1079164
13. Garlan, D., Monroe, R.T., Wile, D.: Acme: Architectural description of component-based systems. In: Leavens, G.T., Sitaraman, M. (eds.) Foundations of Component-Based Systems, pp. 47–68. Cambridge University Press (2000)
14. Hirsch, D., Inverardi, P., Montanari, U.: Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving. In: Donohoe, P. (ed.) Software Architecture (TC2 1st Working IFIP Conf. on Software Architecture, WICSA1). pp. 127–143. Kluwer, San Antonio, Texas, USA (1999)
15. Hirsch, D., Montanari, U.: Consistent transformations for software architecture styles of distributed systems. Electronic Notes in Theoretical Computer Science 28 (2000), http://www.sciencedirect.com/science/article/pii/S1571066105806267
16. Kacem, M.H., Jmaiel, M., Kacem, A.H., Drira, K.: Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. In: ICEIS (3). pp. 189–195 (2005)

17. Kandé, M.M., Strohmeier, A.: Towards a uml profile for software architecture descriptions. In: Proceedings of the 3rd international conference on The unified modeling language: advancing the standard. pp. 513–527. UML'00, Springer-Verlag, Berlin, Heidelberg (2000), http://dl.acm.org/citation.cfm?id=1765175.1765230
18. Kleene, S.C.: On notation for ordinal number. The Journal of Symbolic Logic pp. 150–155 (Dec 1938)
19. Kleene, S.C.: Introduction to metamathematics. Bibl. Matematica, North-Holland, Amsterdam (1952)
20. Le Métayer, D.: Describing software architecture styles using graph grammars. IEEE Trans. Softw. Eng. 24, 521–533 (July 1998)
21. Loulou, I., Kacem, A.H., Jmaiel, M., Drira, K.: Towards a unified graph-based framework for dynamic component-based architectures description in z. Pervasive Services, IEEE/ACS International Conference on 0, 227–234 (2004)
22. Luckham, D.C., Kenney, J.J., Augustin, L.M., Vera, J., Bryan, D., Mann, W.: Specification and analysis of system architecture using rapide. IEEE Trans. Software Eng. 21(4), 336–355 (1995)
23. Magee, J., Kramer, J.: Dynamic structure in software architectures. In: In Proceedings of the Fourth ACM SIGSOFT Symposium on the Foundations of Software Engineering (1996)
24. Medvidovic, N., Rosenblum, D.S., Redmiles, D.F., Robbins, J.E.: Modeling software architectures in the unified modeling language. ACM Trans. Softw. Eng. Methodol. 11, 2–57 (January 2002), http://doi.acm.org/10.1145/504087.504088
25. OMG: Unified Modeling Language Specification 2.0: Superstructure (2005), oMG doc. formal/05-07-04
26. Oquendo, F.: π-method: a model-driven formal method for architecture-centric software engineering. SIGSOFT Softw. Eng. Notes 31, 1–13 (May 2006), http://doi.acm.org/10.1145/1127878.1127885
27. Parr, T., Fisher, K.: Ll(*): the foundation of the antlr parser generator. SIGPLAN Not. 47(6), 425–436 (Jun 2011), http://doi.acm.org/10.1145/2345156.1993548
28. Rodriguez, I.B., Drira, K., Chassot, C., Guennoun, K., Jmaiel, M.: A rule driven approach for architectural self adaptation in collaborative activities using graph grammars. Int. J. Autonomic Comput. 1(3), 226–245 (May 2010), http://dx.doi.org/10.1504/IJAC.2010.033007
29. Roh, S., Kim, K., Jeon, T.: Architecture modeling language based on uml2.0. In: Proceedings of the 11th Asia-Pacific Software Engineering Conference. pp. 663–669. APSEC '04, IEEE Computer Society, Washington, DC, USA (2004), http://dx.doi.org/10.1109/APSEC.2004.32
30. Rozenberg, G. (ed.): Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations. World Scientific (1997)
31. Selonen, P., Xu, J.: Validating uml models against architectural profiles. SIGSOFT Softw. Eng. Notes 28, 58–67 (September 2003), http://doi.acm.org/10.1145/949952.940081
32. Sharrock, R., Monteil, T., Stolf, P., Hagimont, D., Broto, L.: Non-intrusive autonomic approach with self-management policies applied to legacy infrastructures for performance improvements. International Journal of Adaptive, Resilient and Autonomic Systems p. 19 (2010)
33. Wermelinger, M., Fiadeiro, J.L.: A graph transformation approach to software architecture reconfiguration. In: Joint APPLIGRAPH/GETGRATS Workshop on Graph Transformation Systems (GraTra2000). pp. 200–0 (2000)