Enhanced Graph Rewriting Systems for Complex Software Domain

Cédric Eichler, Thierry Monteil, Patricia Stolf, Luigi Alfredo Grieco, Khalil Drira

Software & Systems Modeling manuscript No.
(will be inserted by the editor)

Enhanced Graph Rewriting Systems for Complex Software Domains

Dynamic Software Architecture, Non-Functional Requirements And Correctness By Construction

Cédric Eichler · Thierry Monteil · Patricia Stolf · Alfredo Grieco · Khalil Drira

Received: date / Accepted: date

Abstract Methodologies for *correct by construction reconfigurations* can efficiently solve consistency issues in dynamic software architecture. Graph-based models are appropriate for designing such architectures and methods. At the same time, they may be unfit to characterize a system from a non functional perspective. This stems from efficiency and applicability limitations in handling time-varying characteristics and their related dependencies. In order to lift these restrictions, an extension to graph rewriting systems is proposed herein. The suitability of this approach, as well as the restraints of currently available ones, are illustrated, analysed and experimentally evaluated with reference to a concrete example. This investigation demonstrates that the conceived solution can: (i) express any kind of algebraic dependencies between evolving requirements and properties; (ii) significantly ameliorate the efficiency and scalability of system modifications with respect to classic methodologies; (iii) provide an efficient access to attribute values; (iv) be fruitfully exploited in software management systems; (v) guarantee theoretical properties of a grammar, like its termination.

Researches presented in this paper have been partially funded by the ANR in the context of SOP project ANR-11-INFR-001

C. Eichler · T. Monteil · K. Drira
CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
E-mail: {author_name}@laas.fr

C. Eichler · P. Stolf
IRIT; 118 Route de Narbonne, F-31062 Toulouse, France
E-mail: {author_name}@irit.fr

C. Eichler · T. Monteil · P.Stolf · K. Drira
Univ de Toulouse, UPS F-31400, INSA, F-31400,
UTM, F-31100 Toulouse, France

A. Grieco
Department of Electrical and Information Engineering
Politecnico di Bari,
Via Orabona 4 - 70125, Bari, Italy

Keywords Constrained and attributed rewriting systems · Graph rewriting systems · Non-functional requirements · Dynamic software architecture · Correctness by construction.

1 Introduction

Dynamic software architectures enable adaptation in evolving distributed systems [14, 23]. Their description cannot be limited to a unique static topology, but it has to encompass the entire scope of possible configurations [20]. This scope is characterized by an architectural style, qualifying what is correct and what is not. Once this distinction made, system transformations themselves must be specified to depict their applicability conditions and effects. A crucial undesirable implication of these evolutions is a potential loss of correctness, the system withdrawing from the scope of consistency.

Besides correctness, the system has evolving functional and non-functional requirements, which are tightly linked to its appropriateness or efficiency. For example, configurations can be evaluated with reference to quality of service, energy consumption, and robustness to software or machine breakdowns. These *objectives* are potentially *concurrent*. In fact, deploying more software components or using more machines may ameliorate robustness but worsen energy consumption. The satisfaction of an objective depends on the *properties* of each software component, such as the machine it is deployed on, and the components reachable from it. In turn, those characteristics are *dynamic* and may be *interdependent*. The set of entities accessible through a component of the system, for example, recursively depends on the elements accessible through the components reachable in one hop. Said set is prone to evolve as components are deployed

or terminated.

Hence, modeling a system to ease its management carries two particular aspects which are usually considered separately [38]: correctness and appropriateness with regard to functional and non-functional requirements. These concerns motivate the need for suitable description languages and *formalisms avoiding ambiguities for correct architectural design, management and analysis*.

Formal unambiguous methods are necessary to study the consistency of a system at a given time, i.e., its compliance to an architectural style. Several ways of doing so have been developed in the literature. The most immediate approach, checking the consistency of the system at run-time, may lead to combinatorial explosions and the necessity of roll-backs if it is discovered that the system is in an inconsistent state. To efficiently tackle correctness in the scope of dynamic reconfiguration, correctness by construction [35] through formal approaches have emerged [4, 16, 6]. *Based on formal proofs and reasoning in design-time, they guarantee the correctness of a system, requiring little or no verifications in run-time*. A way to achieve such proofs is to investigate the properties of transformations with regard to consistency preservation, so as to ensure that if a transformation is applicable on a correct configuration its result is another correct configuration.

Modelling dynamic systems with graph-based methodologies has a long tradition [26, 27, 17, 5, 32, 13]. As generic models, graphs may be used to represent a broad range of systems according to diverse architectural views. Graph rewriting techniques allow to elaborate style-based frameworks for the specification of dynamic systems granting correct by construction, style-preserving, evolutions. However, they exhibit restraints critically weakening the possibility of assessing a configuration appropriateness when considering non-simplistic systems.

With reference to a concrete example, this article first highlights limitations of currently available graph based methods in describing system properties and their inter dependencies. The running example, known as *DIET*[1,2] [9], consists in a hierarchical load balancer for dispatching jobs over a distributed infrastructure.

A formal extension of graph rewriting systems is then proposed to lift these shortcomings. The pivotal features of this enhancement are: *mutators*, *admissible relationships* specification, and *constraint oriented* encoding. It is demonstrated that the proposed solution brings three main beneficial advantages with respect to classic graph rewriting approaches.

First, experimental results show that the proposed solution is significantly more efficient and scalable than existing one with regard to attribute modifications.

Second, characteristics of the system can be more efficiently assessed by combining evaluation on demand and/or update on modification. A property can be evaluated whenever its value has to be known. To avoid frequent evaluations, this value can also be kept in memory and updated whenever it changes. The choice between these two options rely on the relatives complexities and frequencies of updates and evaluations.

Third, the model allows to quickly grasp the appropriateness of a configuration, identify objectives that can be ameliorated, and component implying constraints violation. Therefore the management of the system and its evolutions is facilitated.

The rest of the paper is articulated as follows: existing approaches and their main features are illustrated in the next section. The running example, DIET, is presented in Sec. 3. Section 4 introduces the proposed formal extension of classical graphs and graph-grammars related theory. Section 5 exploits this enhanced model to characterize DIET, and demonstrates its fitness for appropriateness evaluation and system management. Experimental results regarding the efficiency and the scalability of the proposed method are presented and discussed in Sec. 6. Finally, Sec. 7 is dedicated to conclusion and outlooks.

2 Related Works

2.1 Language-Based Approaches

Architecture Description Languages (ADL) [30, 2, 15, 29] have have been widely used to model software systems [28, 33, 39]. Thanks to a rigorous syntax and semantic, they allow the definition of architectural entities and relations, as well as the description of the structural and behavioral properties and constraints of a system. *However, such languages usually focus on the description of architectural instances, whereas dynamic aspects have been mildly studied [21].* Darwin [29] and ACME [15] only allow component replication and optional components/connections, respectively. Dynamic-Wright [3] adds evolving capabilities to the language Wright [2], limiting itself to predefined dynamics. The system should have a finite number of configurations and reconfiguration policies known in advance.

[1] Distributed Interactive Engineering Toolbox

[2] Sources and further information are available at http://graal.ens-lyon.fr/DIET

2.2 Model-Based Approaches

General-purpose modeling techniques can provide efficient means for handling dynamism, thanks to the definition of reconfiguration rules driving the evolution on an application in run-time. They furnish very intuitive and visual formal or semi-formal description of structural properties [8]. Designing and describing software models using UML, for example, is a common practice in the software industry. UML provides a standardized definition of system structure and terminology, while facilitating a more consistent and broader understanding of software architecture [36]. Nevertheless, the generic fitness of model-based approaches implies some limitations in describing specific issues like behavioral properties. Therefore, they often require the adoption of ad hoc description languages [37, 39] to map architectural concepts into the visual notation of a model (e.g., UML) [27, 22]. *Moreover, in spite of their wide acceptance, UML-based descriptions appear to lack formal tools for efficiently guaranteeing consistency, due to the inherent semi-formalness of UML.*

2.2.1 Graph-Based Approaches

Among model-based approaches, graph-based methods are appropriate for conceiving correct by construction frameworks. Graphs and graphs rewriting have been successfully applied for modeling structural constraints and properties of a vast range of systems in multiple fields, including software architectures. As a generic model, graphs may be used to represent different architectural views, be it component-based [13], service-based [5], event-oriented, or even human applications [32]. Furthermore, this genericness allows, similarly to approaches combining languages- and model-based solutions, the use of graphs to conduct adaptation in systems described with UML.

Within graph-based approaches, a configuration is represented by a graph and graph rewriting rules can express horizontal or vertical transformations, i.e. reconfigurations or refinements. Architectural styles can be characterized by either a type graph [40, 5] or a graph grammar [18, 17]. The first suffers from the same lack of expressiveness as UML-based methods. Graph grammars offer a generative definition of the scope of correctness, where graph rewriting rules have two distinct values. They intervene in both the characterization of an architectural style as part of a rewriting system and in the specification of consistency preserving reconfiguration rules [17]. *This fitness for designing correct by construction transformations is a key motivation for the adoption of graph grammars as a modeling tool of dynamic software architectures*.

2.2.2 Attributed Graphs

The very first thing to consider with graph-based models is the definition of attributes, representing the basic properties of a system element. The most complex solution, adopted by GROOVE[3], is to consider attributes as special vertexes of the graph [12]. In particular, their domain of definition and operations are defined in the form of a many sorted algebraic signatures [11] SIG, thus viewing attributes as elements of a SIG-algebra [12]. A direct implication is a natural manipulation of attributes using predefined operators and their addition or deletion as regular vertexes of the graph. This modularity does not come without drawbacks. Graph rewriting rules rely on finding graph morphisms, a time-consuming problem. *As a consequence, it first seems inefficient to increase the size of the input graphs.*

In a simpler solution, each elements of the graph, i.e., vertexes and edges, is assigned a list of couples representing attributes along with their domains of definition [32]. To allow attribute modifications, graph transformation environments relying on this model usually allows the specification of changes within or alongside a rule. AGG[4] is a well established graph transformation environment. It possesses in particular an efficient transformation engine that can be used on its own. It supports a large range of verification techniques applied to attributed and typed graph grammars. Attribute modifications can be specified within a rule. GMTE[5] is an other engine that handles graph matchings and transformations. It provides specialized features such as inexact matching and connection instructions. Modification instructions, specified alongside a rule, allow attribute modifications.

In both approaches, a rule can only modify an attribute within its scope, i.e., that appears in the rule. *This leads to an increased number of rule applications.* In particular, this may create a domino effect when changing an attribute recursively impact a chain of interdependent ones.

Variable attributes are usually considered in graph rewriting rules alone. However, it may occur in real systems that the value of an attribute is unknown, due to a lack of information or the postponing of a decision. Consequently, attributes of the conceptual graph modeling the system state at a given time should also be variable.

A novel formalism is presented in the sequel of the paper. It mitigates these restraints and makes graph rewriting systems able to efficiently cope with functional and non-functional requirements in evolving contexts.

[3] http://groove.cs.utwente.nl/
[4] AGG: http://tfs.cs.tu-berlin.de/agg.
[5] GMTE: http://homepages.laas.fr/khalil/GMTE

3 Illustrative Example and Problem Statement

3.1 Distributed Interactive Engineering Toolbox

In order to clarify the issues addressed in this article, a practical example is taken from SysFera-DS[6], an industrial solution for federating and managing hybrid HPC environment. DIET [9] is a hierarchical load balancer for dispatching computational jobs over a distributed infrastructure, like a grid or a cloud. This example is studied with regard to horizontal transformations applied to a component-based view. Its architecture is based on a set of agents: Master Agents (MA) manage pools of computational SErver Deamons (SED) via none, one or several strati of Layer Agents (LA). SEDs can achieve specialized computational services. Communications between agents are driven by the *omniORB* naming service (OMNI). MAs listen to client requests and dispatch them through the architecture to the best SED that can carry out the required service.
This application has been described using class diagrams [37], but, in addition to correction-granting issues, the fact that a LA can manage another LA could not be taken into consideration.

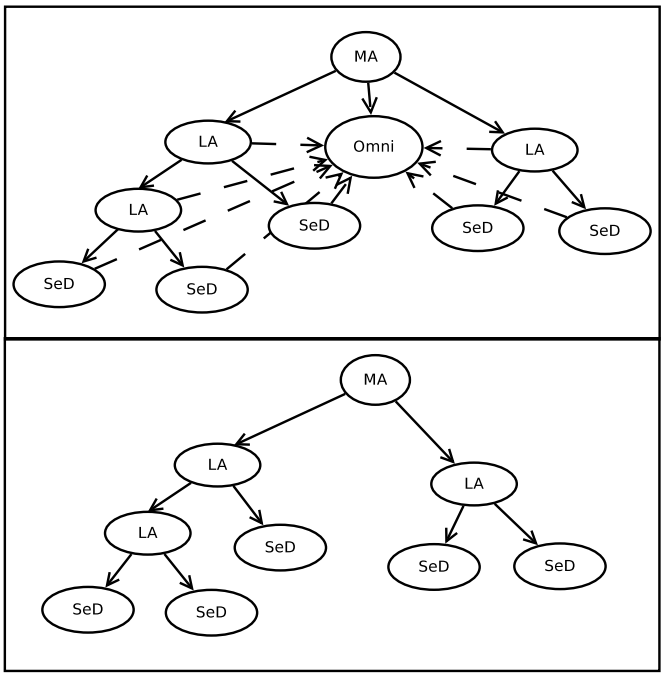

Fig. 1 Logical view of a DIET configuration, with and without an OMNI

Without lack of generality, 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 records itself to the OMNI.
2. Each LA and each SED has a hierarchical superior (i.e., the parent node in the tree).
3. The MA and each LA manage from one (*minSonsMA* / *minSonsLA*) to ten (*maxSonsMA* / *maxSonsMA*) entities. Later in this paper, we will see that these conditions could be trivially extended to any number of minimum and maximum managed entities. Furthermore, from now on, a LA will be said to provide a certain service whenever at least of its child nodes does.
4. Due to hypothetical software restrictions and limited number of machines, the architecture is composed by at most one hundred agents. Once again, this arbitrary value could be expanded to any other.

Figure 1 offers a visual example of how a configuration of DIET may look like, with and without an OMNI. For obvious clarity concerns, the naming service will not be represented in future figures.

All instances of an architectural style are NOT created equal. At a given time, even though a configuration meets all the requirements of the application, another configuration may meet them in a "better way". In particular, we consider the following criteria :

- the energy consumption,
- the robustness, i.e. the fault-tolerance with regard to the breakdown of a machine or a software component, and
- the quality of service.

We assume that the energy consumption depends only on the number of used machines and of the software components deployed on them.
The robustness, instead, is assessed based on three criteria that refers to the set of SEDs running the same service: (i) redundancy degree; (ii) location; (iii) balance within the hierarchical structure. For example, even if multiple SEDs are used for the same service (i.e., redundancy) it is important to allocate them on different machines (i.e., location) to reduce the vulnerability to hardware breakdowns. Similarly, spreading the SEDs far apart within the system tree helps improving the resiliency to LA failures (subject to the constraint that the LAs, not belonging to the same path from the MA to a SED, run on different machines).

Regarding quality of service, the load balance among the different vertice at the same depth in the tree is considered as criterion. 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 a target threshold value, noted maxσ.
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

[6] http://www.sysfera.com/sysfera-ds.html

Table 1 Main Notations for the DIET use-case.

Notation	Meaning
$LA(d)$	the set of LA of depth d
$M(c)$	the number of entities managed by the component c
maxSonsMA *maxSonsLA*	the maximum number of entities managed by the MA or a LA, respectively
minSonsMA *minSonsLA*	the minimum number of entities managed by the MA or a LA, respectively
$\max\sigma$	threshold value of the balancing condition

ameliorating the first, badly impact the second.

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

1. the depth of each LA,
2. the number of entities managed by each component c of type LA and MA : $M(c)$,
3. the set of services carried out by each SED and LA,
4. the machine on which each entity is deployed.

Notations introduced to describe DIET are summarized in Table 1.

3.2 Problem Statement

Herein we illustrate the main issues in modelling the DIET architecture using classical approaches:

3.2.1 Interdependency of attributes

The attributes of an entity v may depend on attributes belonging to a set S of other entities. In classical string grammars, attributes are classified as inherited or synthesized depending on whether the elements of S are parents/siblings of v in the parse tree or not, respectively. The value of synthesized attributes cannot be known in the context where they first appear. They depend on following application of production rules. In graph grammars, these rules traditionally symbolize the addition of software components. Similarly, graph attributes have to be handled in a very different way whether they only depend on attributes belonging to already existing entities or not. The first case can easily be addressed with attributes inheritance in graph grammars. For example, the depth of an LA could easily be derived from the depth of the entity managing it.

This does not apply in the second case. In fact, the set of services offered by an LA, for example, cannot be known in advance since they depends on children nodes that will be added or striped later on. There exist two potential solutions to this problem. Firstly, the attribute may be defined through its analytic expression. This last is either evaluated on demand or systematically re-evaluated after each transformation. Such an evaluation can be time consuming. It may also be unnecessary, for example when the attribute value has not changed. Furthermore, evaluation after each graph transformation is not to be taken lightly. These lasts are not only applied in a deployment step. They also characterize dynamic evolution of the system. Secondly, the attribute may be directly associated to its value and updated when necessary. This last solution is directly related to the modification of an existing attribute discussed in the next sub-subsection.

3.2.2 Modification of an existing attribute

As discussed in 2.2.2, classical approaches allow a graph rewriting rule to modify attributes within its scope only. When considering interdependency of attributes, a modification has to be propagated to dependant attributes. This may lead to a vast number of rule applications. For example, when deploying a SeD on a LA, its set of carried out services have to be updated accordingly. In fact, this update has to be recursively impacted on the ancestor of the updated entity until reaching the MA or a LA that already did carried out each services provided by the new SeD. In this scenario, there are as many rule applications as modified LAs. This phenomena leads to a loss of efficiency and scalability.

3.2.3 Configuration evaluation: handling constraints

Soft and hard constraints can be used to reflect functional and non-functional requirements of a system. Their fulfilment or dissatisfaction enable configuration evaluation.

It is crucial to make the distinction between integrating constraints within the architectural style, building a constrained style, and restraining the architectural style. Existing graph-based approaches are often restricted to the second case, where constraints are used to narrow the scope of correctness only. They are integrated to the model of the style, e.g. to the type graph in [5], but not in the configurations themselves.

These constraints, closely related to the system and its components, are similar to attributes; they depend on attributes, are evolving, and components of the same type have analogous requirements. Hence, their integration in the model as any attributes is relevant. In particular, we wish, while constructing, deploying, or reconfiguring a configuration, to construct an easily evaluable set of constraints. Their violation could be detected and automatically handled by a manager without requiring complex decision and without analyzing the whole application. Firstly, approaches from the literature consider unknown and variable attribute in rules only, but discard their existence from a graph. Thus, classical constraints do not handle such attributes. Secondly, constraints are tackled by post-condition checking or evaluated

after each rule application. Each constraints is then evaluated after a graph transformation even though it may not be changed. This concern is very similar to attribute interdependency.

These three points put under the spotlight the limits of classical graph-based formalism and the need for its expansion described in this paper.

4 Introducing Constraints and Mutators within Graph Rewriting Systems

4.1 Attributes, Constraints and Attributes Rewriting

4.1.1 Attributes

The proposed formalism conserves the simplicity and the computational efficiency of "listing" attributes as labels [32] while granting the possibility of flexibly applying algebraic operators. An attribute is represented as a couple, whose first element represents its value. The second element is its domain of definition. We assume the canonic notation where Y^X is the set of function from X to Y. An interval of integer is noted $[a..b]$.

Definition 1 (Attribute) An attribute is a couple $\text{Att} = (\text{Att}_A, \text{Att}_D)$ where

- Att_A is called value and is either
  - a variable in Att_D,
  - a constant or
  - an expression of a (S, OP)-algebra [12], where (S, OP) is an infinite algebraic signature with S a set of sorts including Att_D and OP a set of function symbols such as $\text{OP} = (\text{Att}_D)^{S^+}$.
- and Att_D is its domain of definition.

An attributed structure or system is a couple composed of the structure and a set of indexed attributes or sequence of attributes. By convention, the first member of an attribute will be noted within quotation marks if and only if its current value is a constant.

4.1.2 Constraints

Attributes are entirely aimed at providing information on an algebraic structure. Constraints can be seen as a specific kind of attributes.

Definition 2 (Constraint) A constraint Cons is an attribute $(\text{Cons}_C, \text{Cons}_D)$ with $\text{Cons}_D = \{$"true", "false", "unknown"$\}$.

Considering that constraints share the same domain of definition, it will be implicit from now on. A constraint $\text{Cons} = (\text{Cons}_C, \text{Cons}_D)$ may be simply referred to as Cons_C. In the following, the principles of Kleene's strong logic [24, 25] are adopted, in particular its basic logic operations $(\vee, \wedge, \neg, \Rightarrow)$ 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 1 A constraint 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 always be evaluated. Any minimal logic expression that can not be evaluated, due for example to an attribute implied in its expression being un-evaluable or variable, is "unknown".

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

Definition 3 (AC-object) An object obj alongside a set of attributes ATT and constraints CONS is called an AC-object and noted $(obj, \text{ATT}, \text{CONS})$.

4.1.3 Attributes Rewriting

One of the issues evoked in section 3 is the fact that attributes are prone to evolve. A reconfiguration may thus impact the attributes of the system, the addition of a SED may for example modify the set of services carried out by some LAs. In the literature, classical string rewriting theory [31] tackles this issue by using mutators. A similar approach is adopted here.

Definition 4 (A mutator on an AC-object) A mutator on an AC-object is an arbitrary algorithm updating the value(s) of none, one or some of its attributes and constraints.

According to this definition, the scope of mutators remains limited to 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.

4.2 Attributed Constrained Graph Modelling a Configuration

4.2.1 Definition

An AC-graph, modeling a software snapshot or configuration at a given time, consists in an AC-couple of two AC-sets

of vertexes and edges where an edge is a couple of vertexes (source, destination). Following the commonly used conventions for standard graphical descriptions, one considers that vertexes represent services or architectural components and edges correspond to their related interdependencies. Note that vertexes, edges and the graph itself are AC-systems. For any set S, the cardinality of S is represented as $|S|$.

Definition 5 (AC-graph) An AC-graph is defined by the system G = (V, E, ATT, CONS) where

- V and E $\subseteq V^2$ correspond to the set of vertexes and edges of the graph respectively,
- ATT (resp. CONS) is a family of sets ATT_{el} (resp. $CONS_{el}$), where *el* is a vertex, an edge or the graph itself. Consequently, ATT (resp. CONS) is 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*.
- For any attributes (A^i_{el}, D^i_{el}), A^i_{el} is a constant, a variable or an expression of a $(S,OP_{el,i})$-algebra where $S = \bigcup_{e\in\{\tilde{e}:ATT_{\tilde{e}}\in ATT\}} \bigcup_{j\in[1..|ATT_e|]} D^j_{\tilde{e}}$ and $OP_{el,i} = (D^i_{el})^{S^+}$.

For any graph (V, E, ATT, CONS), an element $el \in V \cup E$ is said to be attributed (res. constrained) if $ATT_{el} \in$ ATT. The graph is partially attributed and constrained since ATT and CONS are indexed by a *subset* of $V \cup E \cup \{G\}$. In this way, an empty set of attributes or constraint is not required if an element is wished not to be attributed or constrained.

Now that AC-graphs are defined, it is possible to represent a configuration of DIET as presented in section 3.

4.2.2 Modelling a Constrained Configuration of DIET

This subsection is dedicated to the definition of a DIET configuration. Concerns expressed in Sec.3 are mapped into the theoretical concepts previously introduced in this Section. For sake of clarity, before formally introducing architectural styles, we show in Figure 2 what a DIET configuration would look like, once represented using an AC-graph.

Notations are reported in Table 2.

In the case of a DIET architecture :

- *Nat*, the set of possible natures of a software component, is equal to {"*OMNI*", "*MA*", "*LA*", "*SED*"} .
- *Link*, the set of possible relationships between entities, equals {"*ma2la*", "*ma2sed*", "*la2sed*", "*la2la*", "*registered*"}

Red and *Loc*, the redundancy and constraints, are further described in the dedicated paragraph.

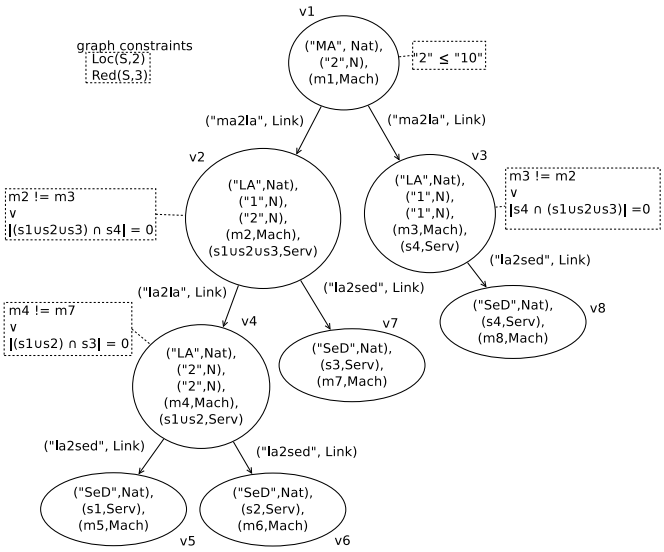

Fig. 2 An AC-graph modeling a configuration of DIET

Table 2 Notations used to describe a DIET configuration (see Fig. 2)

Notation	Meaning
Mach	the set of available machines
Nat	the set of possible natures of a software component
Link	the set of possible relationships
S	the set of services that could be carried out by a SED
Serv	the power set of *S*
Red	the redudancy constraint
Loc	the localisation constraint

Description of the configuration At this time, the software is composed by eight components symbolized by eight vertexes and theirs corresponding relations modeled by some edges, both attributed to reflect their properties and natures. A notable fact is that components of the same nature have the same number of attributes, theirs attributes being the one identified in Sect. 3. This is ensured by the definition of the rewriting system that will be presented later in this paper. Some components as well as the graph itself are constrained to reflect the concerns stated in the same section.

Constraints are represented within doted frames, and related to their targeted object by a doted line, except for those linked to the graph itself.

Attributes The first attributes of each vertex states the nature of the modeled entity, in *Nat*. The configuration comprises a MA managing 2 entities and deployed on a machine noted m_1, as represented by its second and third attribute, respectively.

Each LA possesses three more attributes, related to its depth, the number of entities it managed, the machine it is deployed on and its provided set of services. In the example, three LAs are deployed, represented by v_2, v_3 and v_4 , of depth 1, 1 and 2, managing 2, 1 and 2 entities, and placed on machine m_2, m_3 and m_4, respectively. The first one, v3, manages directly or indirectly SEDs providing the set of services $s_1 \cup s_2 \cup s_3$, the second one, v_4, provides s_4 and v_5, the third one, $s_1 \cup s_2$.
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.
Note that machines and proposed services are represented by variable, and their actual value is not currently know.

Constraints The MA should not manage more that 10 entities, underlining a fundamental property of the architectural style. Load balancing is not represented since it is tackled by conditional deployment, as stated previously.

To cope with robustness, the graph is constrained by two clauses Loc(S,2) and Red(S,3), taking into account the needs for redundancy and multiple locations over the offered services.
$\forall \bar{S} \subseteq S, \forall x_s \in \mathcal{N}$, let the redundancy constraint Red($\bar{S}$,x_s) be "There are at least x_s SEDs carrying each service s in $\bar{S}$".
$\text{Red}(\bar{S},x_s) = \forall s \in \bar{S}, \exists (v_i)_{i\in[1..x_s]} \in V^{x_s}, (\forall (i,j) \in [1..x_s]^2, i \neq j \Rightarrow v_i \neq v_j) \wedge (\forall k \in [1..x_s], \text{ATT}^1_{v_k} = \text{"SED"} \wedge s \in ATT^2_{v_k})$.

$\forall s \in \bar{S}, \forall x_s \in \mathcal{N}$, let the location constraint Loc($\bar{S}$,x_s) be "For each service in $\bar{S}$, there are at least x_s different machines on which at least a SED carrying out the service s is deployed".
$\text{Loc}(\bar{S},x_s) = \forall s \in \bar{S}, \exists (v_i)_{i\in[1..x_s]} \in V^{x_s}, (\forall (i,j) \in [1..x_s]^2, i \neq j \Rightarrow (v_i \neq v_j \wedge ATT^3_{V_i} \neq ATT^3_{V_j})) \wedge (\forall k \in [1..x_s], ATT^1_{v_k} = \text{"SED"} \wedge s \in ATT^2_{V_k}$.

This means that each service should be carried out by at least 3 SEDs located on at least two different machines.

In addition, a notion of location balance within sub-trees is introduced to re-enforce robustness. It is specified that a LA and a component managed by the same entity should not be deployed on the same machine or that they should have disjoint set of carried services. This constraint avoids, within a sub-tree, that SED providing similar services, and deployed in different location thanks to the clause Loc, are managed by entities deployed on the same machine. Hence the number of devices that have to breakdown in order for a service to be disrupted is increased.

Formal definition The graph in the Fig. 2 is defined as follow.

G = (V, E, ATT, CONS) where
$V = \{v_1, v_2, \ldots, v_8\}$,
$E = \{ e_1 = (v_1, v_2), e_2 = (v_1, v_3), e_3 = (v_2, v_4), e_4 = (v_2, v_7), e_5 = (v_3, v_8), e_6 = (v_4, v_5), e_7 = (v_4, v_6)\}$,
$ATT = \{ATT_G, ATT_{v_1}, ATT_{v_2}, \ldots, ATT_{e_8}\}$,

$ATT_{v_1} = \{(\text{"MA"}, \text{Nat}), (\text{"2"}, \mathbb{N}), (m_1,\text{Mach})\}$.
$ATT_{v_2} = \{(\text{"LA"}, \text{Nat}), (\text{"1"}, \mathbb{N}), (\text{"2"}, \mathbb{N}), (m_2, \text{Mach}), (s_1 \cup s_2 \cup s_3, \text{Serv})\}$,
$ATT_{v_3} = \{(\text{"LA"}, \text{Nat}), (\text{"1"}, \mathbb{N}), (\text{"1"}, \mathbb{N}), (m_3, \text{Mach}), (s_4, \text{Serv})\}$,
$ATT_{v_4} = \{(\text{"LA"}, \text{Nat}), (\text{"2"}, \mathbb{N}), (\text{"2"}, \mathbb{N}), (m_4, \text{Mach}), (s_1 \cup s_2, \text{Serv})\}$.
$ATT_{v_5} = \{(\text{"SED"},\text{Nat}), (s_1, \text{Serv}), (m_5, \text{Mach})\}$,
$ATT_{v_6} = \{(\text{"SED"},\text{Nat}), (s_2, \text{Serv}), (m_6, \text{Mach})\}$,
$ATT_{v_7} = \{(\text{"SED"},\text{Nat}), (s_3, \text{Serv}), (m_7, \text{Mach})\}$,
$ATT_{v_8} = \{(\text{"SED"},\text{Nat}), (s_4, \text{Serv}), (m_8, \text{Mach})\}$.

$CONS = \{CONS_G, CONS_{v_1}, CONS_{v_2}, CONS_{v_3}, CONS_{v_4}\}$,
$CONS_G = \{\text{Loc(S,2)}, \text{Red(S,3)}\}$,
$CONS_{v_1} = \{ATT^2_{v_1} \leq 10\}$,
$CONS_{v_2} = \{ATT^4_{v_2} \neq ATT^4_{v_3} \vee (ATT^5_{v_2} \cap ATT^5_{v_3} = \emptyset)\}$,
$CONS_{v_3} = \{ATT^4_{v_3} \neq ATT^4_{v_2} \vee (ATT^5_{v_3} \cap ATT^5_{v_2} = \emptyset)\}$ and
$CONS_{v_4} = \{ATT^4_{v_4} \neq ATT^3_{v_7} \vee (ATT^5_{v_4} \cap ATT^2_{v_7} = \emptyset)\}$.

From now on, notions allowing to characterize the corresponding architectural style are introduced, ensuring in particular that attributes are correctly updated and that components have the required constraints.

4.3 Graph Rewriting Rules and Grammars

An architectural style can be formalized using a graph grammar. The production rules of such systems require to identify sub-structures by the means of homomorphisms. An unattributed graph homomorphism h between two graphs is defined as an injective function f from the set of vertexes of the first one to the set of vertexes of the second graph so that if there is an edge between two vertexes of the first one there is an edge between their image in the second one. By notational abuse, the image of a vertex v by f is noted $h(v)$, the image of an edge (v,v') is noted $h((v,v'))$ instead of $(f(v),f(v'))$, and the image of a subgraph $\tilde{G} = (\tilde{V}, \tilde{E})$ of G is noted $h(\tilde{G})$.

To tackle attributes, we impose firstly that two vertexes or two edges associated through a homomorphism have the same number of attributes. Attributes of two associated elements are themselves correlated with regard to the order of their occurrences. Identified attributes should have the same

domain of definition. Secondly, identifications of attributes should be consistent, e.g. a variable should not be identified with two different constants. Therefore, a system of equations is built and the existence of an attributed induced subgraph isomorphism is conditioned by its resolvability.

Definition 6 (AC-graph homomorphism) A homomorphism between two AC-graphs G = (V, E, ATT, CONS) and G' = (V', E', ATT', CONS'), noted G→G', is a homomorphism h from (V, E) to (V', E') such as

1. $\forall\, el \in V \cup E, |ATT_{el}| = |ATT_{h(el)}|$.
2. $\forall\, el \in V \cup E, \forall\, i \in [1..|ATT_{el}|], D^i_{el} = D^i_{h(el)}$.
3. The system of equations S ={ A = A' : $\exists\, el \in V \cup E, \exists\, i \in [1..|ATT_v|]$, $A = A^i_{el} \wedge A' = A^i_{h(el)}$ } has at least one solution.

Remark 2

- Constraints do not impact the definition of a homomorphism. It will be shown that they intervene in the rewriting process in a different way. Similarly, attributes on vertexes and edges are the only one that are considered whereas attributes on the graph itself are not.
- The existence of a homomorphism is conditioned by the resolvability of a system of equations on attributes. As stated in the introduction, in attributed graphs [19, 12], 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.

Solving the system of equations S results in identifying the value of some attributes with some constants in their domains of definitions and/or with the value of some other attributes. Integrating the affectation obtained by solving the systems refers to the update of the value of the attribute to reflect these identifications. For example, if ((x,y), (x,"2")) $\in S^2$, meaning that x has been identified to the variable y and the constant "2", integrating the affectation obtained by solving S will lead to replacing each occurrence of x and y by "2".

There exists a vast number of approaches handling graph rewriting based on attributed graphs [19, 12]. Their applicability depends on various factors, always including the existence of a homomorphism between an element of the graph rewriting rule and the graph to rewrite. *Inspired by string grammar theory [31], these factors are expanded herein to include the satisfaction of a set of constraints on attributes, namely the set of constraints of the AC-rewriting rule.* This potentially empty set 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 vertexes 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. For any AC-graph $G = (V, E, ATT, CONST)$ and any of its subgraph $\tilde{G} = (\tilde{V}, \tilde{E}, \tilde{ATT}, \tilde{CONST})$, the notation $G \backslash \tilde{G}$ refers to G deprived of $\tilde{G}$, i.e. the graph $\bar{G} = (\bar{V}, \bar{E}, \bar{ATT}, \bar{CONST})$ where :

- $\bar{V} = V \backslash \tilde{V}$,
- $\bar{E} = E \cap \bar{V}^2$,
- $\bar{ATT} = \{ATT_{el} \in ATT : el \in \bar{V} \cup \bar{E}\} \cup ATT_G$,
- $\bar{ATT} = \{ATT_{el} \in ATT : el \in \bar{V} \cup \bar{E}\} \cup ATT_G$.

Virtually, any attributed graph rewriting formalism could be extended to include semantic predicates, constraints and mutators. In order to fix the idea, the classical double push out formalism defined in [34] has been chosen, alongside with the attribute management presented previously.

Definition 7 (AC-rewriting rule of AC-graph) An AC rewriting rule of an AC graph is a 5-tuple (L, K, R, ATT, CONS, ACT) where

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

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

1. there is a homomorphism $h : (L, ATT_L, CONS_L) \rightarrow G$, implying in particular that the system of equations S ={ A = A' : ($\exists$ v ∈ V_L, $\exists$ i ∈ [1..|ATT_v|], $A = A^i_v \wedge A' = A^i_{h(v)}$) ∨ ($\exists$ e = ($\bar{v}, \tilde{v}$) ∈ E, $\exists$ i ∈ [1..|ATT_e|], $A = A^i_e \wedge A' = A^i_{(h(\bar{v}),h(\tilde{v}))}$)} has at least a solution,
2. the application of the rule would not lead to the apparition of any dangling edge,
3. each Cons ∈ $CONS_{rule}$ 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 1.

Its application consists in :

1. erasing h(L\K) including $CONS_{h(L\backslash K)}$,
2. integrating the affectations obtained by solving S to the remaining graph,
3. adding an isomorph copy of R\K, including $CONS_{R\backslash K}$, integrating the affectations obtained by solving S,

4. performing each action Act ∈ ACT.

Graph rewriting rules treat vertex and edge constraints much like attributes. They are added and suppressed alongside the element they target.

Efficient access to attribute values : evaluation on demand or update on modification Note that, thanks to mutators, this formalism enforces several ways of considering and evaluating attributes or constraints. These lasts can be explicitly characterized by their analytic expression. However, this expression has to be calculated whenever its value is required or after each transformation. To avoid frequent evaluations, the attribute value can be stored and be updated whenever it has to be, using mutators. The choice between these two options rely on the relative complexities and frequencies of updates and evaluations.

Inspired from Chomsky's generative grammars [10], graph grammars are defined as a classical grammar or rewriting system, and formally characterize an architectural style.

Definition 8 (**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 a set of AC-vertexes, called non-terminal term of the grammar,
- T is a set of AC-vertexes terminal term, named terminal term of the grammar,
- P is the set of 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.

Terminal terms define archetype of vertexes with corresponding pattern of attributes and constraints. On the other hand, production rules grant constraint management and system updates. Terminal terms and productions guarantee that each component, at any time, of the system is correctly constrained and attributed according to its type.

Definition 9 (**Instance belonging to the graph grammar**) An instance belonging to the graph grammar (AX, NT, T, P) is a graph 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.

Correct-by-Construction Reconfigurations. Correct by construction reconfigurations based on the generative aspect of graph grammars is one of theirs key advantages. A transformation is considered correct if its application to an instance of the grammar produce another one. Productions of the grammar are correct by definition. Thanks to operations on graph rewriting rules that preserve their correctness, correct transformations can be built starting from productions rules. Applicability restriction, for example, is such an operator.

Let r be a rewriting rule whose application is equivalent to the application of a production p. It is immediate that r preserves consistency if its applicability conditions are equivalent to or stronger than those of p, e.g. if r requires a larger pattern to be found meaning that L_r is a sub-graph of L_p. This still holds in presence of mutators and constraints. In addition to classical requirements, the application of two rules is equivalent if they have the same mutators. If the application conditions of a rule are stronger than those of a rule p, they still are if the first is as least as constrained as the second, e.g. if $CONS_p \subseteq CONS_r$.

4.4 Summary of the Proposed Contribution

In the previous Sub-Sections, a complete description of the proposed formalism has been detailed. Here, for sake of clarity, we highlight its pivotal features and advantages in a concise form.

- *Attributes* are enriched to cover their interdependencies and potentially unknown values. Their definition, rather than being restricted to predefined operators and dependencies, is based on the characterization of every admissible relationships.
- *Constraints* are defined as a special kind of attributes, so as to benefit from their evolution and dependencies mechanisms. Being elements of a ternary logic system, they cope with unknown attributes.
- *Graph rewriting rules* are expanded with the consideration of constraints and mutators. Firstly, constraints on the rule itself constitute semantic predicates that allow decision making in presence of unknown attributes. Constraints are added and deleted alongside the element they target. Secondly, mutators, adapted from classical string theory, manage efficiently and flexibly attribute modifications.

Accordingly, it is possible to extend graph grammar *approaches in order to embrace these new features, capitalize their strengths, and enable the effective management of dynamic software architectures subordinate to functional and non-functional requirements.*

5 Exploitation and Illustration of the New Formalism : DIET Characterization, Evaluation and Management

This Section illustrates the potential of the elaborated formalism by first describing DIET, taking into account each consideration introduced in Sect. 3. Then, the fitness of this description to appropriateness evaluation and performance aware management is demonstrated using concrete examples.

5.1 DIET Characterization.

This section is dedicated to the characterization of the DIET application described in Sect. 3 using the new formalism presented in this contribution. To this end, we design axioms, terminal terms, and production rules of the Graph Grammar that unambiguously define DIET. Also, we formally demonstrate the termination of the resulting grammar.

5.1.1 Axiom

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. In the case of DIET, attributes and constraints shared by all possible software configurations are:

1. the largest number of entities that a LA may manage (the minimum being directly granted by production rules),
2. the largest number of entities that a MA may manage (idem),
3. the threshold value intervening in the balancing condition discussed in Sec. 3,
4. the maximum of total agents and
5. the current number of agents.

Common constraints, instead, refers to redundancy and location conditions each configuration has to satisfy.

Therefore, let AX_{DIET} be (v_{AX}, ATT_{AX} = ((maxSonsLA, $\mathbb{N}$), (maxSonsMA, $\mathbb{N}$), (maxσ, $\mathbb{R}^+$), (maxAgents, $\mathbb{N}$), (curAgents, $\mathbb{N}$)), $CONS_{AX}$ = (Loc(S,2),Red(S,3))), where curAgents = 0 and, arbitrarily, maxSonsMA = maxSonsLA = 10 and maxAgents = 100.

Throughout this section, the graph on which production rules will be attempted to be applied to is noted G = (V, E, ATT, CONS). Attribute and constraint inheritance ensure that if G is an instance of the architectural style defined here, $ATT_G = ATT_{AX}$ and $CONS_G \subseteq CONS_{AX}$.

5.1.2 Terminal Terms

These terms characterize types of AC-vertexes, defining a pattern of attributes and constraints shared by vertexes of the same kind.

The naming system itself is not constrained, and its attributes are limited to its nature and the machine it is deployed on. Therefore, 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)), $CONS_{MA}$ = ((Nsons $< A^2_{AX}$))).

Finally, a LA and a component managed by the same entity should not be deployed on the same machine or they should have disjoint set of carried services. Let $\hat{v}$ be the entity managing v, i.e. $\hat{v} \in V$ such that $(\hat{v},v) \in E$, and sib(v) = $\{ \bar{v}:(\hat{v},\bar{v}) \in E_G\} \setminus \{v\}$ the set of components managed by $\hat{v}$, excluding v, i.e. the siblings of v.
T_{LA} is (v_{LA}, ATT_{LA} = ((“LA”, Nat), (depth, $\mathbb{N}$), (Nsons, $\mathbb{N}$), (m, Mach), (s, Serv)), $CONS_{LA}$ = (c(v_{LA})), where
$c(v) = (c(v)_i)_{i \in [1..|sib(v)|]}$, $\forall \tilde{v} \in$ sib(v), $!\exists$ i $\in [1..|sib(v)|]$,
(A^1_v = “LA” $\wedge$ $c(v)_i = (A^4_v \neq A^4_{\tilde{v}}) \vee (A^5_v \cap A^5_{\tilde{v}} = \emptyset)) \vee$
(A^1_v = “SED” $\wedge$ $c(v)_i = (A^4_v \neq A^2_{\tilde{v}}) \vee (A^5_v \cap A^3_{\tilde{v}} = \emptyset)$)

5.1.3 Productions of the Grammar

Production rules of the graph grammar formalize the construction of its instances by defining when and how an entity may be deployed and the consequences of such a deployment.

The first rule (p_1) to define is the initialization 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 registers to the naming service and the current number of agents is updated accordingly.
Let p_1 = (L_{p_1}, K_{p_1}, R_{p_1}, $\emptyset$, $\emptyset$, $\mu_{registering}$(pv2), μ_{inc}(G, 5, 3)), where $\mu_{registering}$(v) is the action of registering the object represented by the vertex v to the naming service. μ_{inc}(e, i, 1), defined in Fig. 3, represent the incrementation of the i-th attribute of v by n.
Graphical parts of the rewriting rules are illustrated here using the format L←K→R. This graphical representation is illustrated in Fig. 4, where L_{p_1}, K_{p_1}, R_{p_1} and pv2 are defined.

Productions rules p_2 and p_3 model the addition of a non-terminal vertex, managed by the MA or a LA, respectively. This temporary vertex will later on be instantiated into a LA or a SED. To deploy a new entity, three condition should be

$\mu_{inc}(e, i, n)$
$A_v^i \leftarrow A_v^i + n$

Fig. 3 μ_{inc}(v, i, n), Incrementation of the i-th attribute of the element e by n

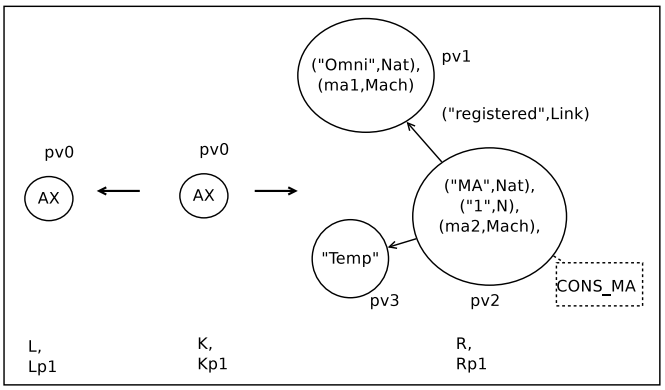

Fig. 4 Initialisation

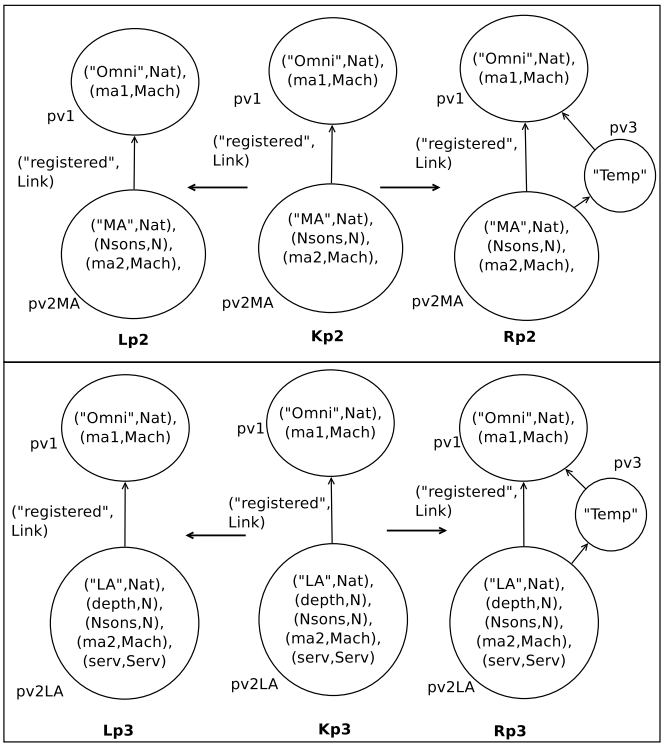

Fig. 5 Addition of a non-terminal term

met. This addition should respect (1) the balancing condition, (2) the maximal number of agents manageable by its superior and (3) the maximum number of total agents. The application of these productions leads to the incrementation of the numbers of total agents and of sons of the entity managing the added vertex.
Let $p_2 = (L_{p_2}, K_{p_2}, R_{p_2}, \emptyset, A_G^4 > A_G^5, (\mu_{inc}$(pv2MA, 2, 1), μ_{inc}(G, 5, 1))) and
$p_3 = (L_{p_3}, K_{p_3}, R_{p_3}, \emptyset, (\text{balancing(pv2LA)}, \text{Nsons} < \text{ATT}_G^1, A_G^4 > A_G^5), (\mu_{inc}$(pv2LA, 3, 1), μ_{inc}(G, 5, 1)),
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. 5. balancing(v) = $\sigma((A_{la}^3)_{la \in LA(A_v^2)\setminus\{v\}}, A_v^3+1) < A_G^3$, where σ(s) is the standard deviation of the sequence s.

The instantiation of a temporary vertex managed by the MA or a LA into a SED is described by p_4 and p_5, respectively. After deploying the SED, it has to register to the naming service and, if it is managed by a LA, update the set of its carried out services. Let $p_4 = (L_{p_4}, K_{p_4}, R_{p_4}, \emptyset, \emptyset, \mu_{registering}(\text{pv4}))$ and $p_5 = (L_{p_5}, K_{p_5}, R_{p_5}, \emptyset, \emptyset, (\mu_{registering}(\text{pv4}), \mu_{updateServ}(\text{pv2,pv4,2}))$, 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. 7. $\mu_{updateServ}$(v, $\tilde{v}$, ind), described in Fig. 6, impact a change in $A_{\tilde{v}}^{ind}$, the set of carried out services by $\tilde{v}$, on v, the component managing $\tilde{v}$, by updating the set of services it proposes. This update is conducted only if v is a LA, and, if there is indeed a change, it is propagated to the entity managing v.

$\mu_{updateServ}(v, \tilde{v}, ind)$
  if A_v^1 = "LA"
    oldServ $\leftarrow A_v^5$
    $A_v^5 \leftarrow \text{oldServ} \cup A_{\tilde{v}}^{ind}$
    if $A_v^5 \neq$ oldServ
      $\tilde{v} \leftarrow \hat{v} \in V_G, (\hat{v}, v) \in V_E$
      $\mu_{updateServ}(\tilde{v}, v)$

Fig. 6 $\mu_{updateServ}$(v, $\tilde{v}$, ind), A change of $A_{\tilde{v}}^{ind}$, the set of services carried out by $\tilde{v}$, impacts v, the entity it is managed by.

The two last productions of the grammar, p_6 and p_7, describe the instantiation of non-terminal term into a LA managed by the MA and a LA, respectively. Since a LA has to manage at least one entity, such an instantiation can be conducted only if an entity can be later on deployed without exceeding the maximum number of agents. Let $p_6 = (L_{p_6}, K_{p_6}, R_{p_6}, \emptyset, A_G^4 > A_G^5, (\mu_{registering}(\text{pv4}), \mu_{inc}(\text{G, 5, 1})))$ and $p_7 = (L_{p_7}, K_{p_7}, R_{p_7}, \emptyset, A_G^4 > A_G^5, (\mu_{registering}(\text{pv4}), \mu_{inc}(\text{G, 5, 1})))$, 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. 8.

5.1.4 The Constrained Attributed Graph Grammar Characterizing DIET

Considering the sets introduced in this section, GRS_{DIET}, the graph rewriting system formally characterizing DIET, introduced in Sect. 3, is defined as GRS_{DIET} =(AX_{DIET}, NT_{DIET}, T_{DIET}, P_{DIET}), where
$\text{NT}_{DIET} = (v_{temp}, \text{ATT}_{temp} = (\text{“temp”}, \{\text{“temp”}\}), \text{CONS}_{temp}$

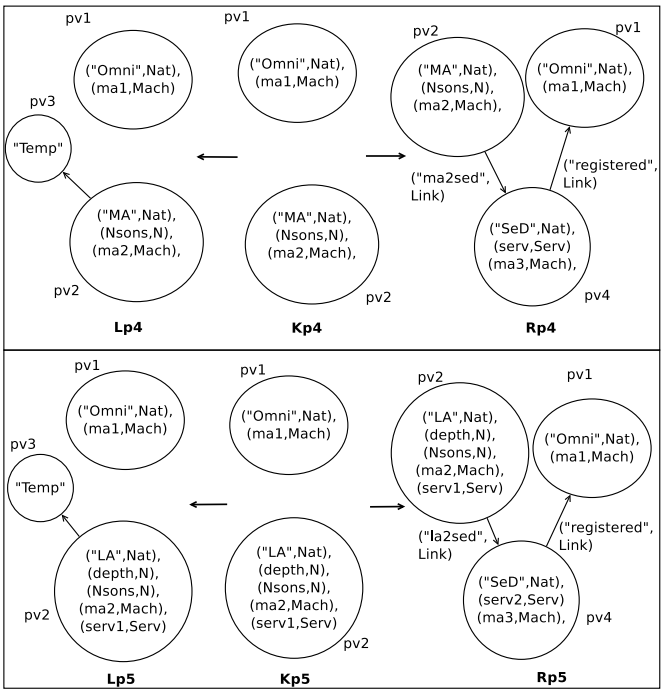

Fig. 7 Instantiation of a non-terminal term into a SED

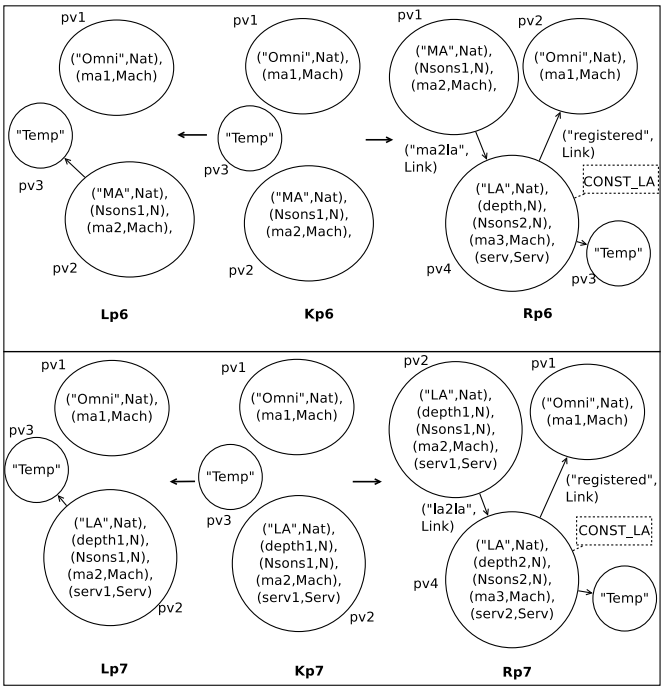

Fig. 8 Instantiation of a non-terminal term into a LA

$= \emptyset$),
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}.

Note that the limitation of entities that can be managed by the MA or a LA are not tackled in the same way. A constraint reflecting this restriction is added on the MA, whereas the satisfaction of this limitation is granted for the LAs by a semantic predicate. Said predicate restricts the applicability of p_3 by imposing, before making a LA manage a new component, that said LA as not reach the limit of component it can manage. Since p_3 is the only production of the grammar increasing the number of entities managed by a LA, this limit can not be overpassed. A brief summary of the mapping between the concerns expressed in Sec. 3 and formal concepts is presented in Table 3.

Loc(S,2),Red(S,3)

$\forall s \in \bar{S}, \forall x_s \in \mathcal{N}$, let the location constraint Loc($\bar{S}$,x_s) be "For each service in $\bar{S}$, there are at least x_s different machines on which at least a SED carrying out the service s is deployed".
$\text{Loc}(\bar{S},x_s) = \forall s \in \bar{S}, \exists (v_i)_{i\in[1..x_s]} \in V^{x_s}$, ($\forall (i,j) \in [1..x_s]^2$, $i \neq j \Rightarrow (v_i \neq v_j \wedge ATT^3_{V_i} \neq ATT^3_{V_j}) \wedge (\forall k \in [1..x_s]$, $ATT^1_{v_k}$ = "SED" $\wedge s \in ATT^2_{V_k}$.

Guaranteeing theoretical properties of the grammar: Termination. A generative grammar is said to be terminating if there can not be an infinite sequence of its production rules. Theoretically, this property ensures that the set of instances of the grammar is finite and that its exploration or the construction of an instance can be represented by a terminating algorithm. Practically, this property is consistent with the finiteness of the available resources, like machines and theirs computing powers.

Theorem 1 *GRS_{DIET} is terminating.*

Proof Let S be a non-empty sequence of elements in P_{DIET} and |S| its size. Let's prove that $\exists$ N $\in \mathbb{N}$, |S| $\leq$ N.

For p $\in P_{DIET}$, let Occ(p) be the number of occurrences of p in S_p. Accordingly,

$$|S| = \Sigma_{p\in P_{DIET}} Occ(p). \quad (1)$$

Let's consider the following system of tokens :

- Token A : G_4 - G_5, the number of agents that still can be deployed.
- Token B : the number of temporary vertexes in the graph.

Applying p_2, p_3, p_6 or p_7 decrease the number of token A by 1, whereas p_1 requires 3. Hence,

$$3 * Occ(p_1) + Occ(p_2) + Occ(p_3) + Occ(p_6) + Occ(p_7) \leq maxAgents. \quad (2)$$

The application of p_4 or p_5 consumes 1 token B, whose number is increased when applying p_1, p_2 or p_3. Hence,

Table 3 Informal considerations and their formal translation

Notation	Formal expression	Description
	Temporary vertexes in R_{p_1}, R_{p_6} and R_{p_7}	The MA and LAs are deployed alongside with *minSonsMA* and *minSonsLA* temporary vertexes, respectively
$CONS_{MA}$	$Nsons < A^2_{AX}$	The MA is constrained not to exceed *maxSonsMA*
$CONS^2_{p_3}$	$Nsons < ATT^1_G$	A new entity cannot be managed by a LA that has reached *maxSonsLA*
$CONS_{p_2}$ $CONS^3_{p_3}$ $CONS_{p_6}$ $CONS_{p_7}$	$A^4_G > A^5_G$	Any transformation eventually implying the deployment of a new component can be applied only if the system is currently composed by less than *maxAgents* entities
$CONS^1_{p_3}$ $= balancing(v)$	$\sigma((A^3_{la})_{la \in LA(A^2_v)\setminus\{v\}}, A^3_v{+}1) < A^3_G$	Incrementing the number of entities managed by v respect the balancing condition
$sib(v)$	$\{\ \bar{v}{:}(\hat{v},\bar{v}) \in E_G\} \setminus \{v\}$ where $\hat{v} \in V$ such that $(\hat{v},v) \in E$	Siblings of v
$CONS_{LA} = c(v)$	$(c(v)_i)_{i\in[1..\lvert sib(v)\rvert]}, \forall \tilde{v} \in sib(v), !\exists\ i \in [1..\lvert sib(v)\rvert]$, $(A^1_v = \text{“LA”} \wedge c(v)_i = (A^4_v \neq A^4_{\tilde{v}}) \vee (A^5_v \cap A^5_{\tilde{v}} = \emptyset)) \vee$ $(A^1_v = \text{“SED”} \wedge c(v)_i = (A^4_v \neq A^2_{\tilde{v}}) \vee (A^5_v \cap A^3_{\tilde{v}} = \emptyset))$	A LA and each of its siblings should not be deployed on the same machine or they should have disjoint set of carried services
$CONS^1_{AX}$ $Red(S,3)$	$Red(\bar{S},x_s) = \forall s \in \bar{S}, \exists (v_i)_{i\in[1..x_s]} \in V^{x_s}, (\forall (i,j) \in [1..x_s]^2,$ $i \neq j \Rightarrow v_i \neq v_j) \wedge (\forall k \in [1..x_s], ATT^1_{v_k} = \text{“SED”} \wedge s \in ATT^2_{v_k})$	Each service is carried out by at least 3 SEDs
$CONS^2_{AX}$ $Loc(S,2)$	$Loc(\bar{S},x_s) = \forall s \in \bar{S}, \exists (v_i)_{i\in[1..x_s]} \in V^{x_s}, (\forall (i,j) \in [1..x_s]^2, i \neq j \Rightarrow$ $(v_i \neq v_j \wedge ATT^3_{V_i} \neq ATT^3_{V_j})) \wedge (\forall k \in [1..x_s], ATT^1_{v_k} = \text{“SED”} \wedge s \in ATT^2_{V_k}$	The set of SEDs offering a service is dispatched on at least 2 different machines

$$Occ(p_4) + Occ(p_5) \leq Occ(p_1) + Occ(p_2) + Occ(p_3). \quad (3)$$

Since p_1 consumes the axiom, it is obvious that

$$Occ(p_1) = 1. \quad (4)$$

Equation (2) thus becomes :

$$Occ(p_2) + Occ(p_3) + Occ(p_6) + Occ(p_7) \leq maxAgents - 3. \quad (5)$$

By definition, $\forall\ p \in P_{DIET}$, $Occ(p) \geq 0$. Accordingly, equations (3), (4) and (5) give

$$Occ(p_4) + Occ(p_5) \leq maxAgents - 2. \quad (6)$$

According to equation (1),
$|S| = Occ(p_1) + (Occ(p_2) + Occ(p_3) + Occ(p_6) + Occ(p_7)) + (Occ(p_4) + Occ(p_5))$.
Thanks to equations (4), (5) and (6), this translates into
$|S| \leq 1 + (maxAgents - 3) + (maxAgents - 2)$.

Finally, $|S| \leq 2{*}maxAgents{-}4$. With $maxAgents = 100$, we have $|S| \leq 196$.

QED.

5.2 Appropriateness Evaluation

To enable the evaluation of DIET configurations, constraints are herein assigned a, potentially infinite, weight. In this way, the appropriateness of a configuration is reflected by calculating its opposite, i.e. the configuration cost. Said cost is calculated as the sum of the costs of its *energy consumption* and of the *violated constraints*. The violation of a constraint in a configuration implies that every defined criteria is not respected. In this case, the configuration is not robust enough and its cost is therefore increased depending on the weight of the violated constraint. A configuration of infinite weight is considered incorrect, so that strong constraints are still enforced.

Notations Let ξ be the function of evaluation; $\forall$ cons $\in$ CONS, $\forall$ c $\in$ cons, $\xi(c) = 1$ if c is “true” and 0 else.

Energy Consumption In Sec. 3, it has been presumed that energy consumption depends on the used machines and the number of deployed components only. In the following, this relation is (realistically according to [7]) assumed to be linear, and weighted by λ_{mach} and λ_{entity} for used machine and deployed component, respectively. Note that the number of current deployed component is already an attribute of the graph. For an easier evaluation, the number of used machines can be added as attribute of the graph as well, and updated whenever necessary, i.e. when applying production

p_1, p_4, p_5, p_6 or p_7. The energy consumed by a configuration is then: $\lambda_{mach} \cdot A^5_G + \lambda_{entity} \cdot |V|$.

Constraint violations It is clear that the constraint reflecting the limitation on the number of entities managed by the MA should not be violated and therefore has an infinite weight. Constraints reflecting the robustness of the system are, however "soft" and are given arbitrary finite weight. The cost of violating the constraint stating that "a LA and a component managed by the same entity should not be deployed on the same machine or they should have disjoint set of carried services", c(v), is weighted by the depth of the LA. Redundancy and location constraints are weighted by λ_R and λ_L respectively.

The cost related to the violation of constraints is :
$\lambda_L\ \xi(\text{CONS}^1_G) + \lambda_R\ \xi(\text{CONS}^2_G) +$
$\lambda_{MA}\ \Sigma_{ma \in V, A^1_{ma} = \text{"}MA\text{"}}\ \xi(\text{CONS}^1_{ma}) +$
$\Sigma_{la \in V, A^1_{la} = \text{"}LA\text{"}}\ \lambda_{LA}\ (\text{A}^2_{la}\ \xi(\text{CONS}^1_{la}))$.

Part of the configuration illustrated in Fig. 2 is arbitrarily instantiated to be totally evaluable and presented in Fig. 9. S, the set of services that may be carried out by a SED, is $\{S_1, S_2, S_3\}$.

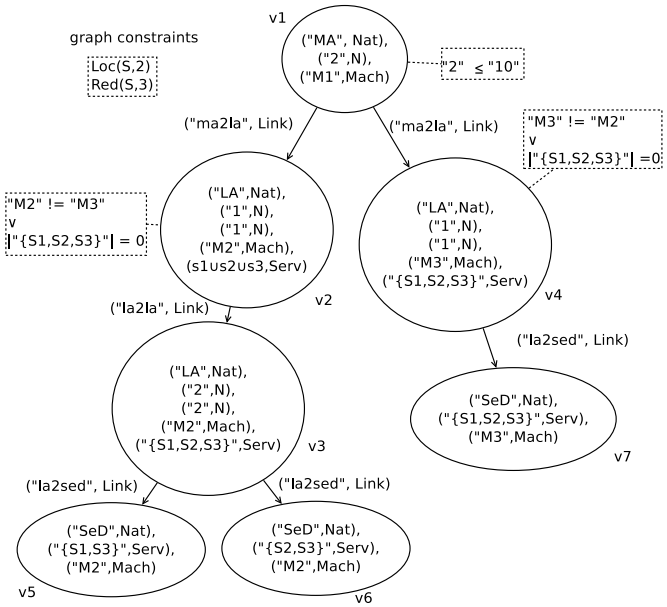

Fig. 9 Instantiated AC-graph modelling a configuration of DIET

This configuration does not meet the redundancy constraint, since there are only two SEDs that can carry out the services S1 and S2. Hence, its cost is equal to its energy consumption plus the cost of violating said constraint: $3\lambda_{mach} + 7\lambda_{entity} + \lambda_R$.

This natural and immediate way to deal with soft and hard requirements derives from the new formalism proposed in this paper. In fact, the model has been explicitly conceived to embed constraints, that may assume the role of performance indicator, and their admissible soft bounds by means of attributes. In this way, once the software architecture properly described, its appropriateness can be easily evaluated on a dynamical basis.

5.3 Non-Functional Requirements and Software Management

Thanks to the eased manipulation of system attributes and constraints, the introduced formalism can *be fruitfully exploited in software management systems*. Semantic predicates and restrictions on rules' applicability grant more flexibility on transformations, allowing to face specific aims on the fly.

Remark 3 Considering a new rule resulting of the restriction of another can ensure guarantees with regard to the preservation of the architectural style. However, it is worth noticing that properties of the graph rewriting system, e.g. confluence, are not necessarily invariant with regard to the addition of a new rule.

Let's consider the DIET configuration previously evaluated (see Fig. 9). In this case, we can suppose that a SED is to be deployed to meet the redundancy constraint and improve the quality of the configuration.

The first thing to do is to apply p_2, and by doing so choosing the component that will manage the SED. To find an optimum solution, one should consider each possibility, i.e. apply p_2 on v_2, v_3 or v_4, find in each case an optimal solution, and then compare the costs of each solutions.
Arbitrarily assuming that p_2 has been applied on v_2, an optimal solution can be found as follows. The temporary vertex is to be instantiated into a SED using the production p_5. Since the motivation of this reconfiguration is to meet the redundancy constraint, p_5 should be restricted in order for its application to be relevant.
Firstly, the deployed SED should be able to provide the services S1 and S2. Assuming the notation introduced when defining p_5, see Fig.7, $\{S1, S2\} \subseteq A^2_{pv4}$.
Furthermore, the constraint ("M2" $!= A^3_{pv4}$) $\vee$("{S1,S2,S3}" $\cap\ A^2_{pv4} = \emptyset$) will appear on v_3. In order for this constraint to be met, the transformation should verify that $A^3_{pv4} \neq$ "M2". Besides, for energy consumption reason and so as not to use a new machine, it is imposed that $A^3_{pv4} \in \{M1, M3\}$.

According to the style constraints, the graph presented in Fig. 10 is a possible optimal result of such a reconfiguration.

Every constraints are met and the cost of the configuration is now limited to its energy consumption, $3\lambda_{mach} + 8\lambda_{entity}$. Hence, this evolution is relevant if and only if $\lambda_{entity} < \lambda_R$.

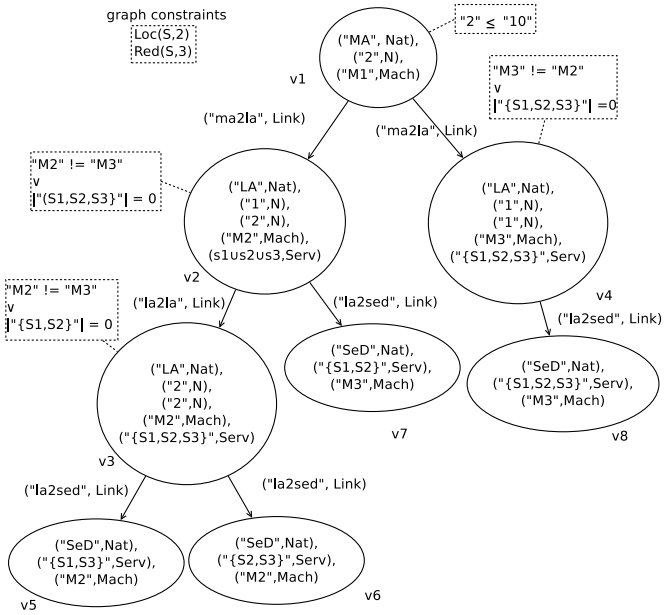

Fig. 10 A configuration satisfying every style-defined constraints

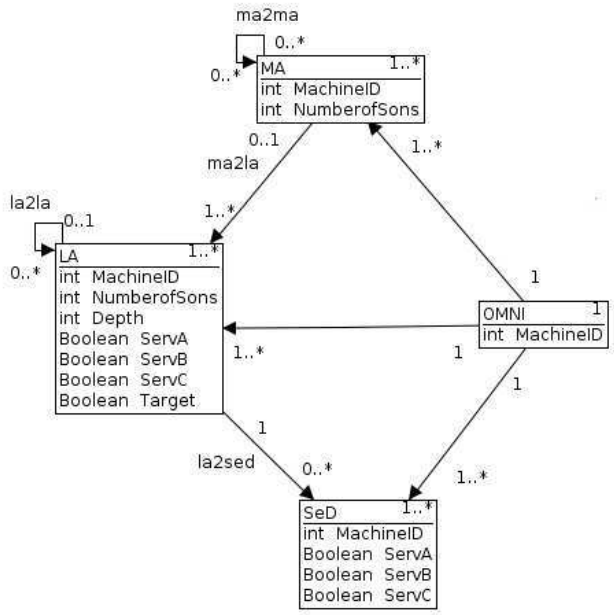

Fig. 11 Experimental DIET style

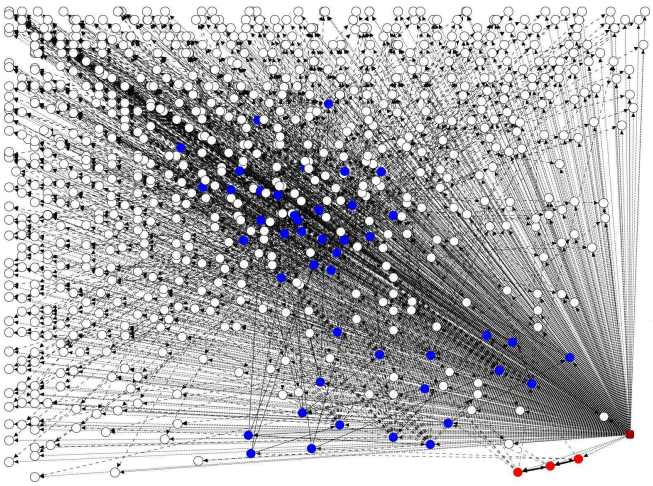

Fig. 12 Smallest experimental graph: A DIET configuration of size 1000 and height 5.

6 Experimentations : Efficiency Evaluation and Comparison

6.1 Experimental Background

6.1.1 Transformational Scenario

The transformation conducted in the experiments consists in the addition of a SeD on a LA of maximal depth. This transformation implies two attribute updates. Firstly, the attribute representing the number of entities managed by the LA has to be incremented. Secondly, the introduction of a SeD may impact the services carried out by its ancestors. The appended SeD provides a formerly un-carried service. As a consequence, the set of services carried out by each of its ancestors has to effectively be modified.

Following the method introduced in this paper, this transformation is a sequential composition of the rules p_3 and p_5 presented in Sec. 5.1.3. Attribute updates are realized through μ_{inc} and $\mu_{updateServ}$.

6.1.2 Manipulated Configurations

In this experimental part, we now consider a non-simplified DIET architecture. The corresponding type graph is represented in Fig. 11. A configuration may contain several connected MAs. As a result, a configuration may exhibit cycles. The manipulated graphs possess the following characteristics :

- Each of them has several MAs and at least one cycle.
- Trees composed by sub-graphs induced by a MA alongside the LAs and SeDs it manages do not have the same height. Within such a tree, SeDs do not necessarily have the same depth.
- There exists no n such as any of these trees is n-ary.
- Each non-intermediary LA manages at least 100 SeDs.

Considered configurations have various sizes and heights. The size of a configuration is the total number of components it is composed by. Its height is the greatest number of LAs between a MA and a SeD. The smallest transformed graph is illustrated in Fig. 12. In this figure, blue and red circle represent LAs and MAs, respectively. The OMNI is illustrated as a red rounded rectangle.

Size and height of graphs significantly influence the scenario execution. In native methods, the height of a graph is equal to the number of rule applications. The complexity of a rule application is strongly related to the size of the graph. In a more superficial way, it does also depend on the topology of the graph and, thus, on its number of MAs. Here, the effect of the graph topology on rule application is not relevant for our study. Therefore, configurations intervening in the experimentations have a fixed number of MAs, set to 3.

6.1.3 Tested Tools and Methods

Experimentations are conducted using two different tools : AGG and GMTE. Both enable the modification of any attribute belonging to the right hand side of a rule. For each engine, experiments are conducted using three methods :

- A referential that does not include attribute modifications. It allows the estimation of time required to carry attribute modifications.
- The native method of the engine. This last consists in a sequence of rule applications. Firstly, the SeD is added to a LA whose set of carried out services is updated. Then, a rule is successively applied to impact this update on each of its ancestors. Each application updates a single LA since modified attributes have to be within the scope of the rule.
- The method proposed in this paper. To this end, we have implemented two overlays relying on AGG and GMTE that carry out the execution of rules p_3 and p_5 with mutators.

Considering two different tools show that the proposed method and the conclusion drawn from experiments are engine-agnostic.

6.2 Experimental Results

Figures 13 and 15 illustrate, for GMTE and AGG respectively, the execution times of the transformational scenario using native methods and mutators. They are represented alongside a referential time computed in absence of attribute modification. Figures 14 and 16 show the evolution of these times for bigger graphs. Illustrated execution times are the median result on 100 executions. Experimentations have been conducted on a computer possessing a quad core processor

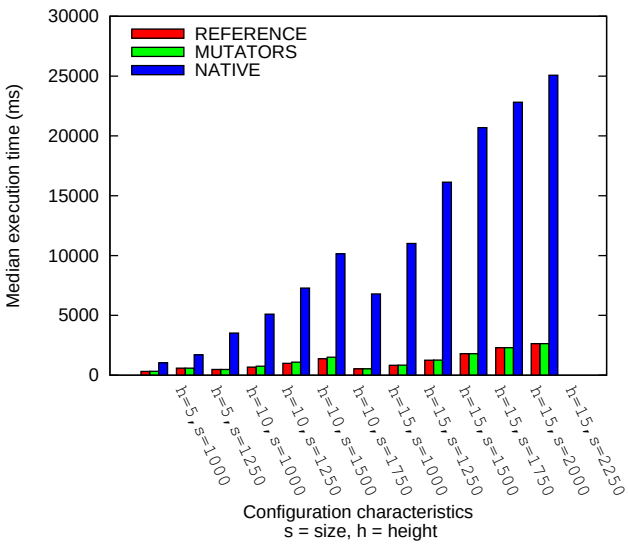

Fig. 13 GMTE : execution times

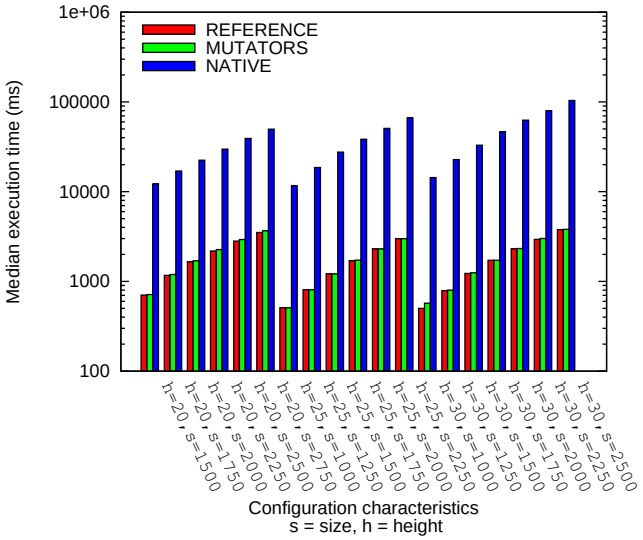

Fig. 14 GMTE : scalability comparison

(4M Cache, 2.66 GHz, 1333 MHz FSB) and 8 Go of RAM. Each configuration is characterized by its size and its height.

Firstly, experimental results show that the overhead implied by attribute modifications through mutators is small in regard of the total transformational time. For example, attribute modifications represent 7 of the 325ms required to conduct the scenario on a configuration of size 1000 and height 5 for GMTE, and 1.2 of the 24.8 ms in the case of AGG. This value increases linearly in height and remains roughly invariant with regard to size. For a configuration of size 2250 and height 30 it amounts to 58 out of 3010ms and 2.7 out of 36.5ms, using GMTE and AGG respectively.

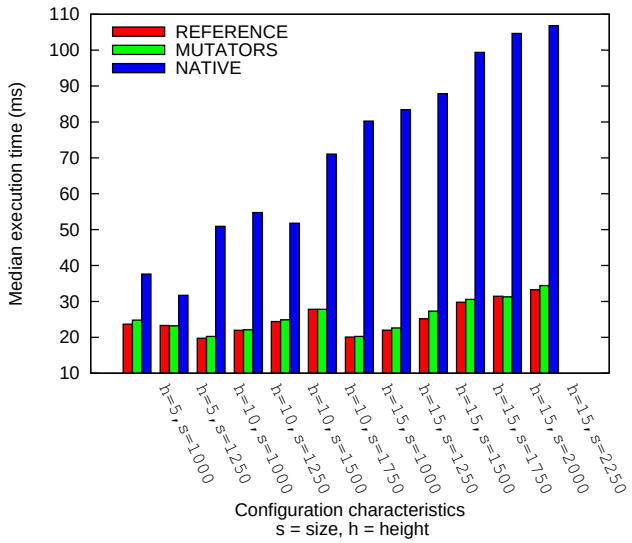

Fig. 15 AGG : execution times

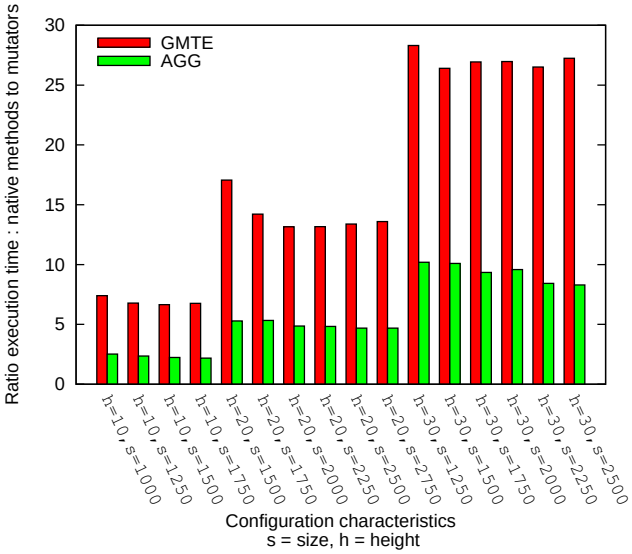

Fig. 17 Native methods vs. mutators : ratio of execution times

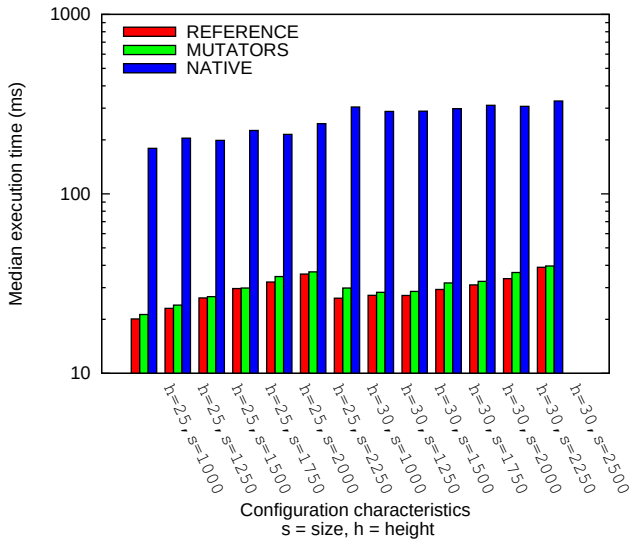

Fig. 16 AGG : scalability comparison

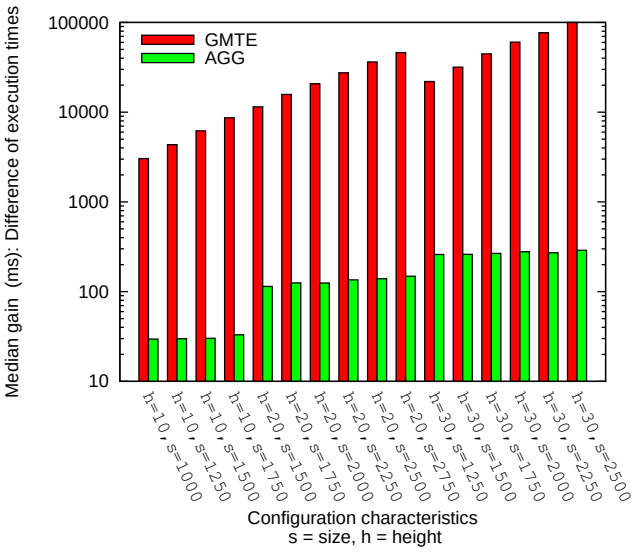

Fig. 18 Native methods vs. mutators : net gain

Secondly, mutators *significantly ameliorate the efficiency and scalability of system modifications with respect to classic methodologies*. Figure 17 depicts the evolution of the the execution time ratio of natives methods to mutators. Considering a configuration of size 1750 and height 20, for example, this ratio is roughly equal to 14 for GMTE (from 16952 to 1192 ms) and 5.3 for AGG (from 154 to 28.9 ms). It increases with height and logarithmically decreases with size. For example, the considered ratio goes up to 27 for the GMTE (from 103796 to 3807 ms) and 8.3 for AGG (from 330 to 39.6 ms) when considering a configuration of height 30 and size 2500. Unlike the considered ratio, the net gain, i.e. the difference of execution times, is strictly increasing with regard to both height and size, as shown in Fig. 18.

These results can be explained by the fact that the proposed method requires a single rule application followed by the execution of a mutator that is linear with regard to depth. The native methods require d rule applications to update attributes within a graph of height d. A rule application itself has a polynomial executional complexity with regard to the size of the graph for AGG, while it is exponential for GMTE. The net gain increases with the application time of a rule.

The results are not independent from the chosen scenario. Here occurs a propagating modification of attributes that is a typical example of the domino effect evoked in

Sec. 2. However, the scenario also comprised the worst comparison case, namely the incrementation of the number of managed components. Since this last impacts attribute within the scope of the rewriting rule solely, mutators are in this case equivalent to classical, native, methods.

7 Conclusion

Dynamic software architectures enable adaptation in evolving distributed systems. They focus on two particular aspects which are usually considered separately: correctness and appropriateness with regard to functional and non-functional requirements. Graph and graph rewriting based methodologies are appropriate for designing correct-by-construction reconfigurations of dynamic systems, effectively guaranteeing their consistency by requiring little or no verification in run-time. Their genericness allows the representation of a vast range of systems in different fields, including dynamic software architectures.

With reference to DIET, an industrial application contributing in federating and managing hybrid HPC environment, this article first shows that currently available graph based methods exhibit limitations in handling varying attributes and constraints. Then, an extension of graph grammars is proposed so as to lift the highlighted restrictions. The pivotal features of this new formalism are: *mutators*, *admissible relationships* specification, and *constraint oriented* encoding. The firsts are introduced within graph rewriting rules as a lightweight approach to *attribute and constraint modifications*. On the other hand, *attribute interdependencies* are expressed through *algebraic operators* that allow to *characterize admissible relationships*. Finally, to ease application management operations, the appropriateness of a configuration in accordance to functional and non-functional requirements is reflected by constraints. Noticeably, to cope also with unknown attributes, constraints are defined as elements of a ternary logic systems.

The application of the resulting formalism to the specification of DIET demonstrates its fitness for the management of systems subordinate to functional and non-functional requirements. Experimental results show that reconfiguring a graph of size 2500 with mutators rather than existing methods is up to 27 times quicker on GMTE and 8.3 times quicker on AGG. This improvement allows to efficiently asses characteristics of the system by combining evaluation on demand and/or update on modification. In turn, this allows to quickly grasp the appropriateness of a configuration, identify objectives that can be ameliorated, and component implying constraints violation.

The initiated extension of AGG is currently being further developed so as to hide the complexity of the formalism within a graphical and user-friendly automated tool. In parallel, mechanisms to take advantage of this new model are being integrated within FRAMESELF [1], a multi-model framework for self-management of distributed systems. Also, ongoing research is exploring the suitability of the proposed formalism to time-constraints. Another interesting future development would be the consideration of infinite logic systems. These lasts provide a more precise way of transforming qualitative properties into quantitative one, by linking, for example, robustness to failure probability.

References

1. Alaya MB, Monteil T (2012) Frameself: A generic context-aware autonomic framework for self-management of distributed systems. In: 21st IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp 60–65
2. Allen R, Garlan D (1997) A formal basis for architectural connection. ACM Transactions on Software Engineering and Methodology 6(3):pp 213–249
3. Allen R, Douence R, Garlan D (1998) Specifying and analyzing dynamic software architectures. In: Fundamental Approaches to Software Engineering, Lecture Notes in Computer Science, vol 1382, Springer Berlin Heidelberg, pp 21–37
4. Baleani M, Ferrari A, Mangeruca L, Sangiovanni-Vincentelli A, Freund U, Schlenker E, Wolff HJ (2005) Correct-by-construction transformations across design environments for model-based embedded software development. In: Conference on Design, Automation and Test in Europe (DATE), pp 1044–1049
5. Baresi L, Heckel R, Thöne S, Varró D (2006) Style-based modeling and refinement of service-oriented architectures. Journal of Software and Systems Modelling 5(2):pp. 187–207
6. Bonakdarpour B, Bozga M, Jaber M, Quilbeuf J, Sifakis J (2010) Automated conflict-free distributed implementation of component-based models. In: International Symposium on Industrial Embedded Systems (SIES), pp 108–117
7. Borgetto D, Casanova H, Da Costa G, Pierson JM (2012) Energy-aware service allocation. Future Gener Comput Syst 28(5):769–779
8. Bradbury JS, Cordy JR, Dingel J, Wermelinger M (2004) A survey of self-management in dynamic software architecture specifications. In: ACM SIGSOFT workshop on Self-managed systems (WOSS), ACM, New York, NY, USA, pp 28–33
9. Caron E, Desprez F (2006) Diet: A scalable toolbox to build network enabled servers on the grid. International

Journal of High Performance Computing Applications 20(3):pp. 335–352
10. Chomsky N (1956) Three models for the description of language. IEEE Transactions on Information Theory 2(3):pp.113–124
11. Ehrig H, Mahr B (1985) Fundamentals of algebraic specification. EATCS Monographs in Theoretical Computer Science. An EATCS Series, Springer-Verlag, Berlin, New York
12. Ehrig H, Ehrig K, Prange U, Taentzer G (2006) Fundamental theory for typed attributed graphs and graph transformation based on adhesive hlr categories. Fundamenta Informaticae 74(1):pp. 31–61
13. Eichler C, Gharbi G, Guermouche N, Monteil T, Stolf P (2013) Graph-based formalism for machine-to-machine self-managed communications. In: 22nd IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp 74–79
14. Gacek C, Lemos R (2006) Architectural description of dependable software systems. In: Besnard D, Gacek C, Jones C (eds) Structure for Dependability: Computer-Based Systems from an Interdisciplinary Perspective, Springer London, pp 127–142
15. Garlan D, Monroe RT, Wile D (2000) Acme: Architectural description of component-based systems. In: Leavens GT, Sitaraman M (eds) Foundations of Component-Based Systems, Cambridge University Press, pp 47–68
16. Gössler G, Graf S, Majster-Cederbaum M, Martens M, Sifakis J (2007) Program analysis and compilation, theory and practice. Springer-Verlag, Berlin, Heidelberg, chap Ensuring properties of interaction systems by construction, pp 201–224
17. Hirsch D, Montanari U (2000) Consistent transformations for software architecture styles of distributed systems. Electronic Notes in Theoretical Computer Science 28(0):pp.4–25
18. Hirsch D, Inverardi P, Montanari U (1999) Modeling software architectures and styles with graph grammars and constraint solving. In: Donohoe P (ed) Software Architecture, The International Federation for Information Processing, vol 12, Springer US, pp 127–143
19. Hoffmann B (2005) Graph transformation with variables. In: Formal Methods in Software and System Modeling, volume 3393 of Lecture Notes in Computer Science, Springer, pp 101–115
20. IEEE (2000) Ieee recommended practice for architectural description of software-intensive systems. IEEE Std 1471-2000 pp 1–23
21. Kacem MH, Jmaiel M, Kacem AH, Drira K (2005) Evaluation and comparison of adl based approaches for the description of dynamic of software architectures. In: 7th International Conference on Enterprise Information Systems (ICEIS), Miami, USA, pp 189–195
22. Kandé MM, Strohmeier A (2000) Towards a uml profile for software architecture descriptions. In: 3rd International Conference on The Unified Modeling Language: advancing the standard (UML), Springer-Verlag, Berlin, Heidelberg, pp 513–527
23. Kephart J, Chess D (2003) The vision of autonomic computing. Computer 36(1):pp 41–50
24. Kleene SC (1938) On notation for ordinal number. The Journal of Symbolic Logic 3(4):150–155
25. Kleene SC (1952) Introduction to metamathematics, Bibliotheca mathematica, vol 1. North-Holland, Amsterdam
26. Le Métayer D (1998) Describing software architecture styles using graph grammars. IEEE Trans Softw Eng 24:pp.521–533
27. Loulou I, Kacem AH, Jmaiel M, Drira K (2004) Towards a unified graph-based framework for dynamic component-based architectures description in z. In: IEEE/ACS International Conference on Pervasive Services (ICPS), IEEE Computer Society, Los Alamitos, CA, USA, pp 227–234
28. Lun L, Chi X (2010) Relationship between testing criteria for architecture configuration testing based on wright specification. In: International Conference on Computational Intelligence and Software Engineering (CiSE), pp 1–4
29. Magee J, Kramer J (1996) Dynamic structure in software architectures. In: 4th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp 3–14
30. Oquendo F (2006) Pi-method: a model-driven formal method for architecture-centric software engineering. ACM SIGSOFT Software Engineering Notes 31:pp. 1–13
31. Parr T, Fisher K (2011) Ll(*): the foundation of the antlr parser generator. SIGPLAN Not 47(6):pp.425–436
32. Rodriguez IB, Drira K, Chassot C, Guennoun K, Jmaiel M (2010) A rule-driven approach for architectural self adaptation in collaborative activities using graph grammars. International Journal of Autonomic Computing (3):pp. 226–245
33. Rong M, Liu C, Zhang G (2011) Modeling aspect-oriented software architecture based on acme. In: 6th International Conference on Computer Science Education (ICCSE), pp 1159–1164
34. Rozenberg G (ed) (1997) Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific
35. Saxena P, Menezes N, Cocchini P, Kirkpatrick DA (2003) The scaling challenge: can correct-by-construction design help? In: International Symposium on Physical Design (ISPD), ACM, New York, NY, USA, pp 51–58

36. Selonen P, Xu J (2003) Validating uml models against architectural profiles. SIGSOFT Software Engineering Notes 28:pp.58–67
37. Sharrock R, Monteil T, Stolf P, Hagimont D, Broto L (2011) Non-intrusive autonomic approach with self-management policies applied to legacy infrastructures for performance improvements. Int J Adapt Resilient Auton Syst 2(1):pp. 58–76
38. Simalatsar A, Guo L, Bozga M, Passerone R (2012) Integration of correct-by-construction bip models into the metroii design space exploration flow. In: 30th IEEE International Conference on Computer Design (ICCD), pp 490–491
39. Tahar BM, Taoufik SR, Mourad K (2013) Checking non-functional properties of uml2.0 components assembly. In: 22nd IEEE International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp 278–283
40. Wermelinger M, Fiadeiro JL (2002) A graph transformation approach to software architecture reconfiguration. Science of Computer Programming 44(2):pp. 133 – 155, special Issue on Applications of Graph Transformations (GRATRA 2000)