Graph-based formalism for Machine-to-Machine self-managed communications

Cédric Eichler, Ghada Gharbi, Nawal Guermouche, Thierry Monteil, Patricia Stolf

Graph-based formalism for Machine-to-Machine self-managed communications

Cédric EICHLER[1,2,3], Ghada GHARBI[1,3], Nawal GUERMOUCHE[1,3], Thierry MONTEIL[1,3]
[1]CNRS; LAAS; 7 avenue du Colonel Roche, F-31077 Toulouse, France
[2]IRIT; 118 Route de Narbonne, F-31062 Toulouse, France
[3]Univ de Toulouse, UPS, INSA, F-31400, UTM, Toulouse, France
{cedric.eichler, ggharbi, nawal.guermouche, thierry.monteil} @laas.fr

***Abstract*— Machine-to-Machine communications comprise a large number of intelligent devices sharing information and making cooperative decisions without any human intervention. To support M2M requirements and applications which are in perpetual evolution, many standards are designed, updated and rendered obsolete. Among these, arise from The European Telecommunications Standards Institute (ETSI) a promising standard for M2M communications. The ETSI M2M provides in particular a standardized framework for interoperable M2M Services. As most of its peer, this standard does not, however, address the issue of dynamic reconfiguration or provide a suitable model for the reasoning required to build self-managed M2M architectures. In our paper, we propose a graph-based approach built on top of the ETSI standard, including rules for reconfiguration management, to enforce self-management properties of M2M communications.**

Keywords-autonomic computing; dynamic reconfiguration; graph model; ETSI M2M Architecture

I. INTRODUCTION

During the last years, the exponential expansion of wireless communications devices and the ubiquity of wireless communications networks have convey to the emanation of wireless Machine-to-Machine (M2M) communications as the most promising solutions to revolutionize the future "intelligent" pervasive communications.

Intrinsically, M2M systems are evolution prone as applications are stopped and started; machines discovered and shut down, etc. As most of its peer, the ETSI standard focuses on protocols and communications and does not addressed the issue of dynamic reconfiguration or provide a suitable model for the reasoning required to build self-managed M2M architectures. These considerations belong to the field of dynamic software architectures studied for handling adaptation in autonomic distributed systems, coping with new requirements, new environments, and failures. We propose in this paper a formal, component-based, bi-layered framework for modelling M2M systems. Our approach is composed by a graph-based layer suitable for reasoning and handling dynamism on the top of the ETSI standard qualifying the behavioural properties of the system. We propose as well generic policies of reconfiguration, relying on graph rewriting, to enforce self-management properties.

The remainder of the paper is organized as follow: Section 2 introduces a state of the art of the Machine-to-Machine communications paradigm, the ETSI M2M architectural model, and model-based approach for the description of dynamic software architectures. In Section 3, after introducing general concepts of graph rewriting systems, we present our approach. Section 4 is dedicated to conclusion and perspectives.

II. STATE OF THE ART

A. Machine-to Machine communications

The Machine-to-Machine (M2M) paradigm is a broad label that can be used to describe any technology that enables automated, wired or electronic devices. The idea of machine-to-machine communications is to enable M2M components to be interconnected, networked and controlled remotely with low cost, scalable and reliable technologies.

M2M communications can be deployed in many applications with the objective of enhancing efficiency and reducing human intervention. Some examples of M2M usages: Automotive, Smart metering, eHealth.

Panday and al. [1] identifies some common characteristics for all M2M applications and based on these ones, they propose a common set of management functionalities for M2M systems. As M2M applications common characteristics, we mention M2M devices' intelligence, heterogeneous networks, two way communication, network dynamics, mobility and time sensibility of data.

We note that the M2M communications are becoming more and more complex. So, the cost of development, maintenance and research in M2M systems are increasing. To meet these challenges, the standardization is a key enabler to remove the technical barriers and ensures interoperable M2M services and networks. Many standards bodies [2, 3, 4, 5] are moving rapidly to support M2M communications requirements. They are working in defining architecture and services standards for M2M applications.

The European Telecommunications Standards Institute (ETSI) [6] has developed an end-to-end architecture for machine-to-machine communications. The ETSI technical committee publishes a various technical reports [7] defining: functional and behavioral requirements of each network element to provide an end-t-end view; the functional architecture with the different M2M services capabilities; the protocols of various interfaces. Only the ETSI M2M group provides an end-to-end view of the global M2M architecture. In addition, technical reports are

available. For these reasons, we have chosen the ETSI specification as a reference to model M2M systems.

B. *ETSI Architecture Model*

The ETSI M2M standardization provides M2M architecture [8] with a generic set of capabilities for M2M services. The ETSI standards facilitate the deployment of vertical applications and the innovation across industries by exposing data and providing services.

ETSI has divided M2M systems, as shown in figure1, into three domains: application domain, network domain and M2M device domain. An M2M device domain includes data end points such as sensors, smart meters, microprocessors, etc. An M2M network domain is any network technology providing connectivity between M2M devices connected to the same M2M area network or allowing an M2M device to gain access to a public network via a router or a gateway. The network domain could be based on standards including PLC, Zigbee, IEEE 802.15, etc. Finally, the M2M application domain runs the service logic and use M2M services capabilities accessible via an open interface. Examples of M2M applications include utilities responsible for collecting and analyzing smart meter data. The application data is referred as resources. Resources are defined in a tree structure and handled with the RESTful style of data exchange.

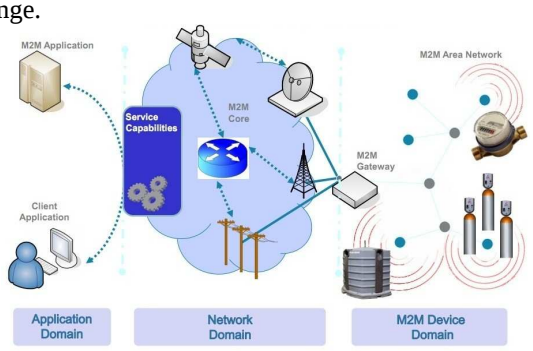

Figure1: ETSI M2M Architecture

In M2M systems, the network server manages an important number of distributed and diverse data. These data comes from a large number of devices and are exchanged between various entities (applications) through Data Containers. These containers are used as a mediator that takes care of buffering the data. They make the exchange abstracted from the need to set direct connections and allow for scenarios where both parties in the exchange are note online at the same time. To accomplish the interaction between the distributed applications and devices (sensors, gateways, etc), a certain number of steps have to be achieved: the registration and the announcement of resources. These procedures are necessary for M2M resource to become known and start exchanging the data.

C. *Dynamic softwares architectures*

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 characterizes an architectural style, qualifying what is correct and what is not. Naturally, once this distinction made, the question of specification of the modifications themselves arise.

Model-based approaches, proposing general-purpose modeling languages, allows to handle dynamism and particularly the definition of reconfiguration rules managing the evolution on an application in run-time. They provide very intuitive and visual formal or semi-formal description of structural properties [9]. For example, designing and describing software models using UML [10] 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 [11]. Nevertheless the generic fitness of model-based approaches implies a poor means of describing specific issues like behavioral properties. Therefore, they are often coupled with description using architecture description languages [12, 13, 14], mapping the concepts of architecture description languages into the visual notation of UML, or other formalism [15, 16].

Among these formalisms, graph-based methods for software modelling are appropriate for conceiving correct by design frameworks, as theoretical work in this field provides formal means to specify and check structural constraints and properties [17, 18, 19]. Within this kind of approaches, some methods are restricted to the usage of type graphs alone [20] and suffer from a lack of expressiveness. Other works [21, 22] are based on graph grammar, or graph rewriting system, and 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 of a graph rewriting system 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.

III. Approach

This section describes the principle and design of the approach proposed in this paper, focusing on the formal layer. It should be stressed out that we do not intend to duplicate the information contained in the ETSI M2M architecture representation, but rather to represent only the relevant one to enforce considered properties through self-management. We first introduce general concepts related to graph rewriting systems, and then the generic formal layer, based on meta-graph grammar, on top of the

standard. Lastly, generic policies for self-management are included.

A. Graph rewriting rule and graph rewriting systems

At a given time, the system is called a configuration or instance of its architectural style and is represented by an attributed graph, whose vertices model entities, such as devices, applications and containers, and edges represent their relationships such as “deployed” or “write on”. Attributes may either be constant or variable and are represented alongside with their domains of definition.

A **graph** G is thus 3-tuple (V, E, ATT) where:

- V and E $\subseteq V^2$ correspond respectively to the set of vertices and edges of G,
- ATT is a family of set indexed by V $\cup$ E. A set of this family is a sequence of couple (A, D_A) where A is either a constant in D_A, noted between quotations marks, or a variable that may take any value in D_A.

An architectural style can be formalized using a graph rewriting system. The productions rules of such systems require identifying sub-structures by the mean of morphisms. An unattributed induced sub-graph isomorphism 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 images in the second one and reciprocally, see [17]. There is an induced sub-graph isomorphism i between two attributed graphs G= (V, E, ATT) and G’= (V’, E’, ATT’) noted – G $\rightarrow$G’- if and only if there is an unattributed **induced sub-graph isomorphism** from (V, E) to (V’, E’) such as:

- $\forall$ v $\in$ V (resp $\forall$ e =($\bar{v}$, $\tilde{v}$) $\in E^2$), $|ATT_v| = |ATT_{h(v)}|$ (resp. $|ATT_e| = |ATT_{h(\bar{v}),\, h(\tilde{v})}|$), (1)
- $\forall$ v $\in$ V (resp $\forall$ e =($\bar{v}$, $\tilde{v}$) $\in E^2$), $\forall$ i $\in$ [1, $|ATT_v|$] , $D^i_v = D^i_{h(v)}$, (2)
- The system of equation S = {A = A’ | ($\exists$ 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. (3)

Solving the system of equation S results in identifying the value of some attributes with some constants in theirs 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, see [23] for more information about these integrations.

Additionally, for genericness sake, we define **super patterns** as:

- a vertex whose only attribute is “any”, its domain of definition begin of no interest. Its attributes does not take part in the conditions (1), (2) or (3). Such a vertex only relevant in the phase where an unattributed sub-graph isomorphism is looked for.
- an attribute taking value in a subset of its domain of definition, materialized by enumerating the possibility, e.g. (“a” or “b”, {“a”, “b”, “c”}). Such an attribute impacts the condition (3) by adding a constraint on the system of equation S.

The characterization of graph rewriting rules used in this paper is based on the Double PushOut (DPO) [24] approach. A **graph rewriting rule** is consequently a 3-tuple (L, K, R) where L and R are two graphs, and K -called the Inv zone- is a sub-graph of both L and R. L\K is called the Del zone and R\K is called the Add zone. A rule is applicable on a graph G if there is an induced sub-graph isomorphism i: L $\rightarrow$G and its application does not lead to the apparition of any dangling edge. Its application consists in erasing (L\K) and adding an isomorph copy of R\K integrating the affectation obtained by solving the system of equations related to i. In this paper, graph rewriting rules are illustrated 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.

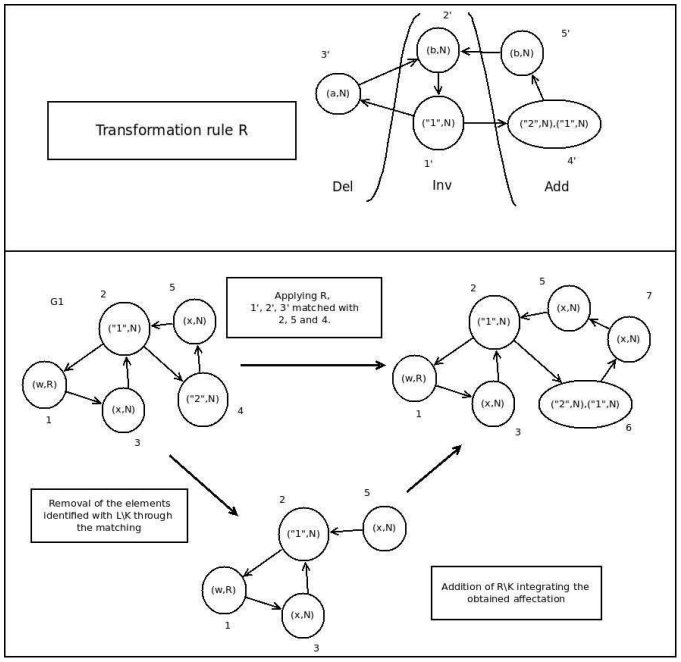

Figure2: an example of graph transformation

Figure 2 offers an example of how transformation is handled in the previously defined approach as well as an illustration of the delta representation. To lighten the figure, the attributes of the edges have not been represented and will be all considered equals. The Del zone, for example, is composed by one vertex noted 3’ and two edges (1’, 3’) and (3’, 2’). Concerning its applicability, considering that there exists an induced sub-

graph isomorphism iso such as L → G1 such as ∀ v ∈ V_{G1} \ iso(V_K), ∀ v' ∈ V_L \ V_K, (v, iso(v')) ∉ E_{G1} ^ (iso(v'), v) ∉ E_{G1}, the deletion of the graph identified with Del through iso would not lead to the apparition of any dangling edge. The transformation R can be applied to G1 with the matching iso. The image of the Del zone is removed and an isomorph copy of the Add zone is then added.

Inspired from Chomsky's generative grammars [25], **graph grammars** are defined as classical system <*AX; NT; T; P*>, where *AX* is the axiom, *NT* is the set of the non-terminal vertices, *T* is the set of terminal vertices, and *P* is the set of graph rewriting rules, also called grammars productions. An instance belonging to the graph grammar is a graph G such as there is not any nt∈*NT* such as nt→ G and the obtained starting from axiom AX by applying a sequence of productions in *P*.

B. The bi-layered approach

The formal layer built to reason and manage actual M2M applications is composed by a generic graph grammar. Said applications are instances of the ETSI standard for M2M architecture. In a similar fashion, management of actual M2M architectures shall rely on instances of the meta-graph grammar.

For conciseness, the information considered here are restricted to:

- The deployed devices, the kind of applications they may run, and whether they are announced or not. When two devices "see each other", i.e. they are announced to one another, the propagation delay due to the physical network through which they communicate.
- The deployed containers, on which device, and whether they are registered or announced.
- The deployed applications, on which device, their type, whether they are registered or announced, and the containers they currently use.

Consequently, the generic graph grammar is <*AX, Ø, T, P*> where:

T = {N((id,Id),(deviceType, {"Network", "Gateway", "ETSIdevice"}), (runnableApplication, applicationTypes)), N((id,Id)), N((id,Id),(applicationType, applicationTypes))}

And *P*= {p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9, p_{10}}. For readability sake and considering that there is no ambiguity on domains of definition, they are implicit in the following.

The production p_1 represents the initialization and the deployment of the "Network" node and is illustrated in figure 3.

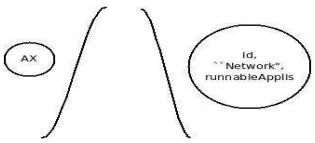

Figure 3: Initialization

The rule p_2 illustrated in figure 4 represents the addition of a device.

Figure 4: Addition of a device

The rule p_3 illustrated in the figure 5 represents the addition and the registration of an application.

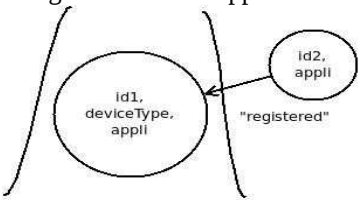

Figure 5: Addition and registration of an application

Figure 6 illustrates the productions p_4 modelling the deployment and registration of a container.

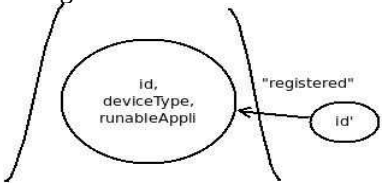

Figure 6: Addition and registration of a container

The rule p_5 presented in the figure 7 formalizes the announcement of a device to the network or a gateway.

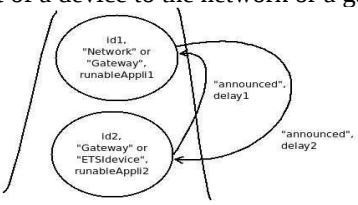

Figure 7: Announcement of a device

The announcement of an application or a container requires the device it is deployed on to be announced as shown by the production p_6 depicted in figure 8.

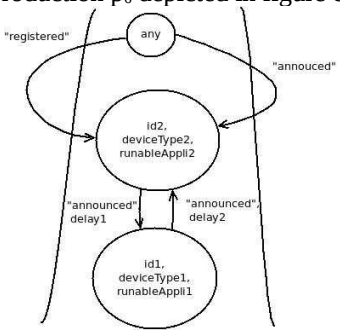

Figure 8: Announcement of an application or a container

An application may use a container, i.e. reads and/ or writes on it, if:

- Both are deployed on the same device, depicted by figure 9,
- The container is on an entity on which the application is announced, p_8 in figure 10, or
- The application is running on an entity on which the container is announced, p_9 in figure 11.

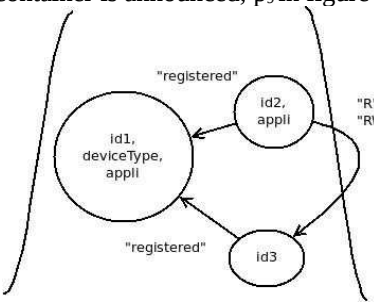

Figure 9: An application use a local container

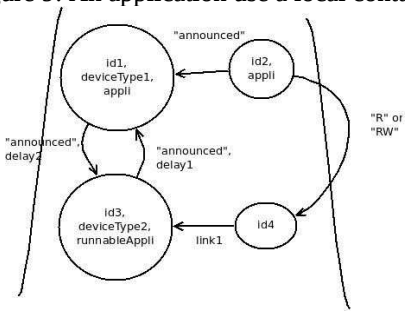

Figure 10: An application use a distant container

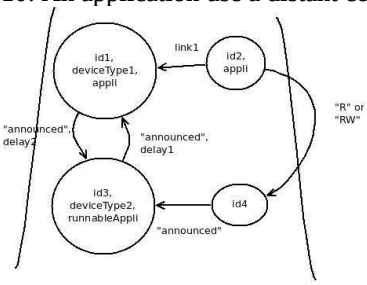

Figure 11: An application use a distant container

Communication between layers is bi-directional. Discovery of a new device on the functional layer triggers the application of the production p_2 with the ad-hoc attributes followed by p_3 and p_4 as many time as necessary, i.e. once by respectively applications and containers registered on the discovered device. On the other hand, when a production is applied consequently to a decision in the formal layer the implication must be impacted. This includes the effective deployment of entities in the "real" world and the necessary calls for registration or announcement on the functional layer. These bi-directional updates ensure the coherence between the two layers. Decisions making in the formal layer and generic algorithms for enforcing self-managed policies are presented in the next sub-section.

C. Enforcement of self-management policies

We now suppose the existence of a monitoring and/or an analysing routine able to throw the following events:

- there is less than x% of battery left on a device d,
- there is less than x% of disk space left on a device d,
- a container c has been accessed more than x times by distant applications in an interval of time t,
- an application of a certain type is needed to be seen from a device d.

Each event triggers an algorithm as described below. The graph representing the formal layer at the time the actions are made is noted G = (V, E, ATT).

"**A container c has been accessed more than x times by distant applications in an interval of time t"**, and should thus be moved to the network in order not to saturate the communication channel of the device where c is deployed. Every application that reads and/or writes on c are redirected to the corresponding container. These actions are described in the algorithm *migrate(idC, idD),* where idC is the identifier of c and idD the identifier of the device where the new container shall be deployed, in this case the Network.

migrate(idC, idD)
createNannouce(idC, idD)
for each induced sub-graph isomorphism i : Lredirect(idc) → G
apply graph rewriting rule redirect(idc, idNewC) w.r.t. i
update the resource tree of the application identified by i
apply graph rewriting rule destroy(idC)
update the resource tree of the device where c used to be deployed.

With *createNannonce(idC, idD)* being the process creating a new container on the device identified by idD, and making every announcement so that each application using the container identified by idC may use the new container.

createNannounce(idC, idD)
apply graph rewriting rule p_4 with id fixed idD
idNewC ← id', the id of the new container
for each induced sub-graph isomorphism i : $L_{announceD(idC, idNewC)} \rightarrow G$
apply graph rewriting rule announceD(idc, idNewC) w.r.t. i
update the resource tree of the device identified by i and the Network resource tree
apply graph rewriting rule p_6 with the isomorphism associating the super vertex with the new container.
Deploy the corresponding container and update the resource trees.

where redirect(idC, idNewC), announceD(idC, idNewC), and destroy(idc) are defined respectively in figure 12, 13 and 14. Note that the uniqueness of the induced sub-graph isomorphism, with regard to which p_4,

p_6, duplicate(idC, idNewC) and destroy(idC) are applied, is ensured by the uniqueness of the identifier of the container.

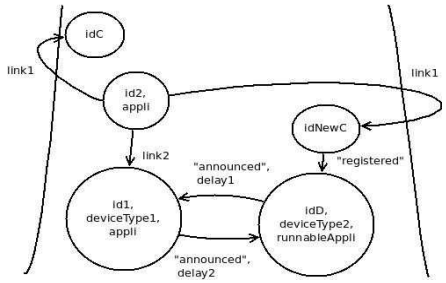

Figure 12: Redirection of an input and/ or output of an application

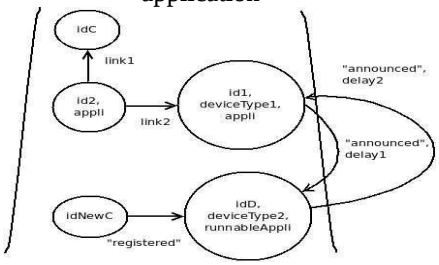

Figure 13: Announcement of a device on which an application to be redirected is deployed

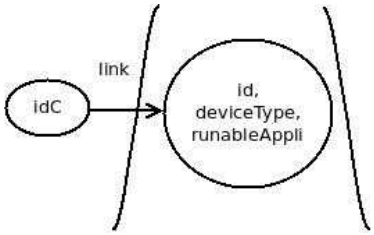

Figure 14: Suppression of the original container

"**There is less than x% of battery left on a device d"**, may lead to the loss of data in the containers deployed on the device d whenever it will shut down due to an empty battery. In order to prevent this loss, each container deployed on d is moved elsewhere and every application that reads and/ or writes on a migrated container is redirected to the corresponding container, as conducted by the process backup(idD).

backup(idD)
for each induced sub-graph isomorphism i : G'(idD) → G
idC ← the identifier of the container associated with id through i.
idTargD ← findSuitableDevice(idC)
migrate(idC, idTargD)

With G' being defined in figure 15 is nothing more than a container deployed on device. findSuitableDevice is introduced at the end of this subsection.

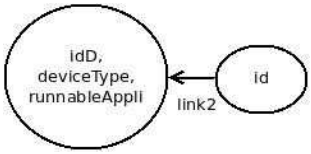

Figure 15: A container registered or announced on a Device

"**There is less than x% of disk space left on a device d",** could lead to the saturation of the disk. Therefore, a container $\sim c$ is deployed elsewhere for each container c deployed on d. Applications writing on c are made to write on $\sim c$ instead, updating the necessary resource trees. Applications reading on c are made to read both on c and $\sim c$.

saturation(idD)
for each induced sub-graph isomorphism i : G'(idD) → G
idC ← the identifier of the container associated with id through i.
idTardD ← findSuitableDevice(idC)
createNannounce(idC, idTargD)
idNewC ← the id of the new container
for each induced sub-graph isomorphism j: $L_{connectReader(idC)}$ → G
apply graph rewriting rule connectReader(idC, idNewC) w.r.t. j
update the resource tree of the application identified by j.
for each induced sub-graph isomorphism j: $L_{connectWriter(idC)}$ → G
apply graph rewriting rule connectWriter(idC, idNewC) w.r.t. j
update the resource tree of the application identified by j

where connectReader(idC, idNewC) and connectWriter(idC,idNewC) are the graph rewriting rules respectively defined in figure 16 and 17.

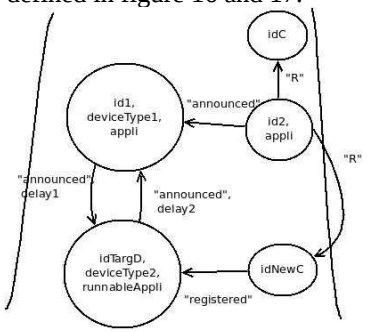

Figure 16: An application can also now read the new container

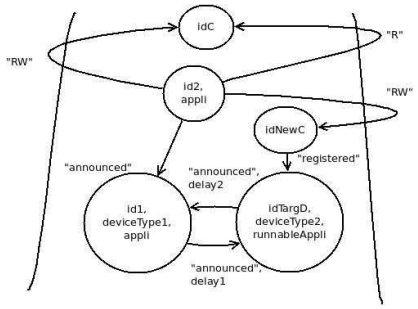

Figure 17: An application can read the old container and read and write in the new one

"**An application of a certain type is needed to be seen from a device d"**, in this case the first thing to do is

to look for such an application and conduct the required announcements. If there is none, such an application shall be started on a device that can run this kind of application. If there is none, such a device shall be deployed. Finally the required announcements are conducted.

lookup(type)
if there is no induced sub-graph isomorphism i : ({N((id,Id), (type, applicationTypes))}, Ø;}) → G
if p_3 is not applicable to G with appli being fixed to type
apply p_2 to G with runnableAppli fixed to type
apply p_3 to G with appli fixed to type
idA ← the attribute identified with id through I or the identifier of the new application
if applicable to G apply announceDevice(idA, idD)
apply announceApp(idA, idD) to G

Where respectively defined in figure 18 and figure 19.

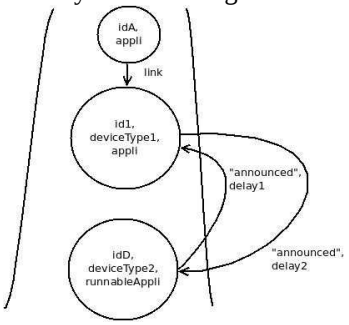

Figure 18: Required announcement of the device

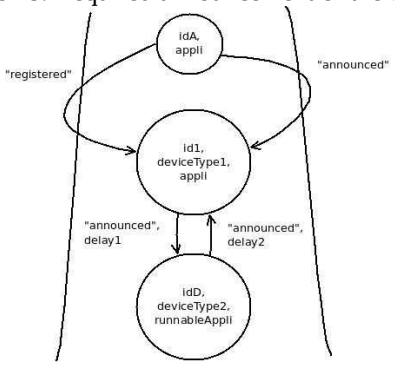

Figure 19: Required announcement of the application

The function **findSuitableDevice(idC)** returns the identifier of the device on which a container ϵ will be deployed in order to replace or reinforce the container c identified by idC. This implies that each application using c will use ϵ. We consider that the suitable device is the one minimizing the sum of the transmission delays from each application to said device. Besides, the potential targeted devices are restricted to the ones on which an application using c is deployed except the one where c is deployed, plus the network. Said set can be constructed by looking for induced sub-graph isomorphisms from R_{p9}, R_{p10}, and R_{p11} to G. It is supposed to be known and noted potential(idC), while potential_id(idC) qualifies its set of identifiers. To compute the transmission delays, we rely on Floyd-Warshall, a well-known algorithm of graph theory solving the all-pair shortest paths, with edges weighted by their attributes "delay". The function FW(G) takes a graph G and returns a function shortest, shortest (id1, id2) being the weight of the shortest path from the vertex identified by id1 to the one identified by id2. If there is no such path, the weight is infinite.

findSuitableDevice(idC)
searchGraph ← the sub-graph of G induced by potential(idC)
shortest ← FW(searchGraph)
for each i ∈ potential_id(idC)
sumFrom(i) ← $\Sigma_{id \in potentialD\text{-}id \setminus \{i\}}$ shortest(i, id)
idD ← id such as sumFrom(id) = $mini \in_{potentailD\_id}$ sumFrom(id)
if sumFrom(idD) is finite return idD
else return idNetwork

Graph rewriting rules used by the presented algorithms are connected to the rules of the graph grammars. Actually, the application of most of them is equivalent to the application of a production or a sequence of productions of the grammar. They only differ in their applicability conditions by requiring larger patterns to be found. The suppression of a container forms a notable exception, and is based on the reversibility of production. These facts ensure that the system stays in a state buildable with a sequence of productions, and thus the correctness of the reconfigurations.

IV. Conclusion

We introduce in this paper a bi-layered approach, benefiting from the best of worlds, for modelling M2M architectures. The ETSI M2M standardization is chosen to describe functional properties and guarantee interoperability between machines. On top of said description, we defined a graph-based generic framework ad-hoc for reasoning and handling the inherent dynamism of M2M architectures. Additionally, we have shown how this formalism may be used to enforce correct generic self-management policies of reconfiguration by defining scenarii and procedures affecting both layers to cope with new requirements and/or prevent failures. Finally, we illustrated the appropriateness of graphs and graph algorithms for decision making.

In the future, we plan on fully defining the interactions between layers that are but evoked in this paper. Defining a set of actions to be performed depending on type of added or removed entities of a graph, or events in the functional layer, would open the path to a formal proof of inter-consistency and, therefore, implementation.

References

[1] S. Pandey, M-S. Mup, M-H. C, and J W. Hong, "Towards Management of Machine to Machine Networks," Network operations and Management Symposium (APNOMS), 2011 13th Asia-Pacific, vol., no., pp.1-7, 21-23 Sept. 2011

[2] "TIA", http://tiaonline.org/standards

[3] "ENSA", http://www.ensa.org/

[4] IP-for the connection of smart objects "IPSO", http://www.ipso-alliance.org/

[5] “Open Mobile Alliance- M2M working group”, Ileana Leuca, ETSI M2M Workshop, Sophia Antipolis, 19-20 October 2010 http://docbox.etsi.org/Workshop/2010/201010_M2MWORKSHOP/06_M2MGlobalCollaboration/LEUCA_OMABOARDx.pdf

[6] “ETSI M2M”, http://www.etsi.org/Website/Technologies/M2M.aspx.

[7] “ETSI Technical reports”, http://www.etsi.org/technologies-clusters/technologies/m2m

[8] “ETSI M2M functional architecture technical”, report http://www.etsi.org/deliver/etsi_ts/102600_102699/102690/01.01.01_60/ts_102690v010101p.pdf

[9] J.S. Bradbury, J.R. Cordy, J. Dingel, M, Werlinger, “A survey of self-management in dynamic software architecture specifications”, Proceedings of the 1st ACM SIGSOFT workshop on Self-managed systems, WOSS’ 04, ACM, New York, USA, 2004, pp 28-33

[10] OMG, Unified Modeling Language Specification 2.0: Superstructure, oMG doc. formal/05-07-04 (2005).

[11] P. Selonen, J. Xu, Validating uml models against architectural profiles, SIGSOFT Softw. Eng. Notes 28 (2003) 58–67. doi:http://doi.acm.org/10.1145/949952.940081. URL http://doi.acm.org/10.1145/949952.940081

[12] S. Roh, K. Kim, T. Jeon, Architecture modeling language based on uml2.0, in: Proceedings of the 11th Asia-Pacific Software Engineering Conference, APSEC ’04, IEEE Computer Society, Washington, DC, USA, 2004, pp. 663–669. doi:http://dx.doi.org/10.1109/APSEC.2004.32. URL http://dx.doi.org/10.1109/APSEC.2004.32

[13] N. Medvidovic, D. S. Rosenblum, D. F. Redmiles, J. E. Robbins,Modeling software architectures in the unified modelinglanguage, ACM Trans. Softw. Eng. Methodol. 11 (2002) 2–57.doi:http://doi.acm.org/10.1145/504087.504088. URL http://doi.acm.org/10.1145/504087.504088

[14] L. Broto, D. Hagimont, P. Stolf, N. de Palma, S. Temate, Autonomic management policy specification in tune, in: ACM Symposium on Applied Computing, Fortaleza, Ceara, Brazil, 2008, pp. 1658–1663.

[15] I. Loulou, A. H. Kacem, M. Jmaiel, K. Drira, Towards a unified graphbased framework for dynamic component-based architectures description in z, Pervasive Services, IEEE/ACS International Conference on 0 (2004)227234.doi:http://doi.ieeecomputersociety.org/10.1109/PERSER.2004.33.

[16] M. M. Kand´e, A. Strohmeier, Towards a uml profile for software architecture descriptions, in: Proceedings of the 3rd international conference on The unified modeling language: advancing the standard, UML’00, Springer-Verlag, Berlin, Heidelberg, 2000, pp. 513–527. URL http://dl.acm.org/citation.cfm?id=1765175.1765230

[17] G. Rozenberg (Ed.), Handbook of Graph Grammars and Computing by Graph Transformations, Volume 1: Foundations, World Scientific, 1997.

[18] H. Ehrig, H.-J. Kreowski, 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.

[19] R. Bruni, A. Bucchiarone, S. Gnesi, D. Hirsch, A. Lluch Lafuente, Graphbased design and analysis of dynamic software architectures, in: P. Degano, R. Nicola, J. Meseguer (Eds.), Concurrency, Graphs and Models, Vol. 5065 of Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2008 pp. 37–56. doi:10.1007/978-3-540-68679-8 4. URL http://dx.doi.org/10.1007/978-3-540-68679-84

[20] M. Wermelinger, J. L. Fiadeiro, A graph transformation approach to software architecture reconfiguration, in: Joint APPLIGRAPH/GETGRATSWorkshop on Graph Transformation Systems (GraTra2000, 2000, pp. 200–0).

[21] D. Hirsch, P. Inverardi, U. Montanari, Modeling Software Architectures and Styles with Graph Grammars and Constraint Solving, in: P.Donohoe (Ed.), Software Architecture (TC2 1st Working IFIP Conf. on Software Architecture, WICSA1), Kluwer, San Antonio, Texas, USA, 1999, pp. 127–143.

[22] D. Le M´etayer, Describing software architecture styles using graph grammars, IEEE Trans. Softw. Eng. 24 (1998) 521–533. doi:10.1109/32.708567.

[23] C. Chassot, K. Guennoun, K. Drira, F. Armando, E. Exposito, A. Lozes, Towards autonomous management of qos through model-driven adaptability in communication-centric systems, ITSSA 2 (3) (2006) 255–264.

[24] H. Ehrig, Tutorial introduction to the algebraic approach of graph grammars, in: H. Ehrig, M. Nagl, G. Rozenberg, A. Rosenfeld (Eds.), Graph- Grammars and Their Application to Computer Science, Vol. 291 of Lecture Notes in Computer Science, Springer Berlin / Heidelberg, 1987, pp. 1–14, 10.1007/3-540-18771-5-40.

[25] N. Chomsky, Three models for the description of language, Information Theory, IEEE Transactions on 2 (3) (1956) 113–124. URL http://ieeexplore.ieee.org/xpls/abs all.jsp?arnumber=1056813