Safe Composition in Middleware for the Internet of Things

Ines Sarray, Annie Ressouche, Daniel Gaffé, Jean-Yves Tigli, Stéphane Lavirotte

Safe Composition in Middleware for the Internet of Things

Ines Sarray
Inria SAM
2004 route des lucioles
06902 Sophia Antipolis,
France
ines.sarray@inria.fr

Annie Ressouche
Inria SAM
2004 route des lucioles
06902 Sophia Antipolis,
France
annie.ressouche@inria.fr

Daniel Gaffé
LEAT, Nice Sophia Antipolis
University and CNRS
930 route des colles
06903 Sophia Antipolis,
France
daniel.gaffe@unice.fr

Jean-Yves Tigli
I3S , Nice Sophia Antipolis
University and CNRS
930 route des colles
06903 Sophia Antipolis
jean-yves.tigli@unice.fr

Stéphane Lavirotte
I3S, Nice Sophia Antipolis
University and CNRS
930 route des colles
06903 Sophia Antipolis
stephane.lavirotte@unice.fr

ABSTRACT

The Internet of Things (IoT) connects sensors, actuators and autonomous objects interacting with each other. These devices are represented by web services. Web services composition often involves conflicts between systems having access to shared devices. In our component-based middleware, our solution allows managing access to shared devices, by generating specific constraint components which guarantee the respect of some predefined composition and adaptation constraints. IoT environments are dynamic; our solution ensures adaptation to its changes by using new generated constraint components and inhibitors to deal with the appearance and disappearance of devices/applications. The main contribution in this work is the definition of a new language DCL (Description Constraint Language) that helps to generate our constraint components by describing generic constraints that must be verified on accesses to shared devices. The whole approach and its associated tools rely on the synchronous paradigm, since it has a well-established formal foundation allowing automatic proofs, and interface with most model-checkers. We can then prove and guarantee a safe composition at runtime for our IoT applications.

Categories and Subject Descriptors

F.1.1 [**Computation by Abstract Devices**]: Models of Computation

Keywords

Middleware, Composition, Validation, Synchronous modeling

1. INTRODUCTION

1.1 Composition of multiple software applications in an IoT context and a physical environment

IoT is a way to combine computation and communication capabilities, sometimes in large scale information systems, with a huge number of complex devices connected to the physical world. Such infrastructures are often dedicated to the deployment of multiple applications, running concurrently. These applications are using shared devices from a common environment through different network middleware and numerous IoT protocols. One of the first challenge to cope with is the devices technological heterogeneity. Today, Web of Things (WoT) (WS-REST [6] and WS-SOAP attached to device) federates most of the protocols and middleware paradigms. Each device then provides Web Services in a global and distributed services oriented approach. Different software models can then be used to describe services compositions, such as those based on process execution languages or component oriented approaches, with various patterns of communication, etc. So, what is new compared with the traditional distributed software approach?

In fact, the real world is intrinsically dynamic and more or less unpredictable. IoT consists in a set of physical devices interacting with a real environment and implies multiple actors with their activities changing over time. Consequently, everything is dynamic and continuous adaptation and reconfigurations of the system are then required.

If the semantic consistency of each application is guaranteed separately by design, the main difficulty is to investigate how this consistency evolves once the applications are running simultaneously. For instance, switching on and off the same light from two applications, drastically changes the consistency of the overall system. A step further, when an application starts the heating and, at the same time, another the air conditioning in a same room, the result of the composition is not what it was expected at design time. Now, the composition of multiple applications is not only based on the interactions between explicit software applications but also influenced by implicit interactions in the real world. To take into account such problematic, concerns and relevant level of abstraction must be chosen for a useful and partial representation of the real world. Thus we define a context that can be for instance operational, physical, social or user-based according to chosen points of views and concepts, according to the context definition of A.Dey

[5]. The "Things" of IoT are just behind the devices. Indeed, “Things”, also called the *Entities of Interest* [7], are the part of the real world in which devices are interacting and which must not be neglected. Thus the model of the system is described by Figure 1.

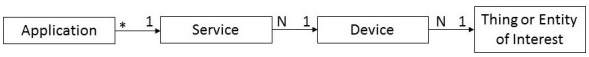

Figure 1: Our Model of a WoT System

We investigate how to model and validate concurrent accesses to shared devices without neglecting their associated *Entity of Interest*, their common physical context. One of the main challenge is then how to guarantee and validate some safety and integrity properties throughout the system's evolution. In our middleware, we use synchronous models to facilitate the study and the validation of new composition mechanisms between applications at runtime. Then key problems to solve are: (1) how to specify and respect the "Thing" behavior? (2) if several services accessing a same entry of an *Entity of Interest*, how to ensure a safe combination of these multiple accesses? (3) Applications may simultaneously use a same service, how to manage these multiple uses?

1.2 Our Contribution for Validation and Composition at runtime

In this paper we address these problems by relying on formal method to model device behaviors as synchronous automata, taking into consideration their impact on the Entity Of Interest. Such an approach allows applying model-checking techniques to verify safety properties of applications. The main contribution is the definition of a sound way to compose models allowing context change adaptation. This composition relies on synchronous parallel composition paradigm. We prove that this operation preserves safety properties. However, it is not sufficient to obtain a global model of this composition because some devices may interact with the same Entity Of Interest. Moreover, several applications may use the same device services and then they can have concurrent accesses to their entries, so, it can have an unexpected impact on our Entity Of Interest. Therefore, we add constraints to the device models composition and to applications level. The major contribution is the definition of a generic way to express these constraints, independently of the knowledge about the devices and the applications, only their type is sufficient. As a consequence, this approach ensures the adaptation to a context change and offers a means to formally perform validation.

This paper is organized as follow: the next section describes a motivating use-case. Then, section 3 introduces the synchronous paradigm on which we rely to ensure safety. Section 4 describes how we model services and applications and details our answer to the context change problem. Section 5 shows the application of our approach to the WCOMP middleware, using the CLEM synchronous toolkit. Section 6 presents previous works addressing middleware reliability problem before concluding in section 7.

2. USE CASE

We introduce a use case which will help us to illustrate our purpose all along this paper. This use case is about the management of a temperature of a room in a flat.The *Entity of Interest* in this example is the temperature controlled room. The temperature is controlled by two internet objects: an air conditioner and a heater. Thus these devices are used by a set of applications. Two applications are available: an application to cool the room (APP_1) and the other to warm it (APP_2). Each application and device is defined by a type and a set of constraints. Paul and Pierre are handling one of these applications using their smartphones.

APP_1 and APP_2 can be activated simultaneously as well as the devices. Thus, the problem of concurrent access can appear. We cannot warm and cool the room at the same time. These applications must respect several constraints: (1) APP_1 is only used by Paul smartphone; (2) APP_2 is only used by Pierre smartphone; and (3) The air conditioner and the heater cannot be activated simultaneously. The question is: how can we manage this conflict and respect these constraints?

This use case is very simple, it is introduced to illustrate our purpose. Nevertheless, our approach is relevant even for this example because we need to rely on a formal modeling to exhaustively ensure that the air conditioner and the heater are not ON simultaneously.

3. THE SYNCHRONOUS MODEL

3.1 Introduction to the Synchronous paradigm

Our goal is to model services as components representing device behaviors. These components are reactive systems: their inputs are actions performed on physical devices and their outputs represent enabled services. This class of systems fits well with the synchronous approach based on the notion of a *logical time*: time is considered as a sequence of logical discrete *instants*.

An instant is a point in time where external input events can be observed, along with the internal events that are a consequence of the latter. In this approach we can model an activity according to a logical time framing: the activity is characterized by a set of events expected at each logical instant and by their expected consequences. A synchronous system evolves only at these instants and is "frozen" otherwise (nothing changes between instants). At each logical instant, all events are instantaneously broadcasted to all parts of the system whose reaction to these events contributes to the global system state.

Each instant is triggered by input events (the core information completed with the internal state computed from instantaneous broadcast performed during the instant frame). As a consequence, inputs and resulting outputs all occur simultaneously. This (ideal) *synchrony hypothesis* is the main characteristics of the synchronous paradigm. Another major feature is also that it supports concurrency through a deterministic parallel composition. The synchronous paradigm is now well established, it relies on a rigorous semantics, and tools have been designed for simulation, verification and code generation. To express these models as Mealy machines [10] offers both design and validation facilities.

3.2 Mealy Machine Representation

Mealy machines are both finite automata and synchronous models. The interfaces of such machines are composed of input events and output events. Each logical instant (or reaction) of the synchronous model is represented by a transition in the machine. Indeed, the considered Mealy machines are composed of : a finite set of states, an initial set and a transition relation between states. Transitions are of the form: $q \xrightarrow{i/o} q'$, where q and q' are states. They bear labels falling into two parts: a trigger part (i) to activate the transition and an output part composed of output events (o). Indeed i is a Boolean composition of some input events and o is a set of output events. Actually, this definition introduces explicit Mealy machine representation as finite state machines (FSM). However, Mealy machines have been introduced by G. Mealy [10] to synthesize sequential efficient circuits as Boolean equation sys-

tems computing both the output event values and the next states from input event values and current states[1]. We call these representations "implicit" Mealy machines. Explicit representation is more design convenient while implicit representation is more convenient for composing machines and validating their properties; mainly because Boolean equation systems support a very compact encoding into Binary Decision Diagram (BDD).

Synchronous models of the expected behaviors of the devices will be provided as well as some additional properties (constraints) that must be respected when these devices are connected to an *Entity of Interest*. These models are designed as Mealy machines where each output is connected with an input event of the *Entity of Interest*. To compose two Mealy machines we define their synchronous product. It is also a Mealy machine where the state space is the product of each machine space state. Each transition between two states is the composition of transitions in the original machines. In this composition, the trigger part is the conjunction of the respective trigger part of the original labels and the output part is the union of the respective output event sets of each transition.

3.3 Validation for Synchronous Models

A major benefit of relying on synchronous models is the easiness to describe explicit Mealy machines and the ability to perform validation. Among the validation techniques, the model-checking [9] requires system models against which formulas are checked for satisfaction. The model must express all the possible system behaviors, the formulas depicted required properties of such behaviors. Mealy machines are well suited to represent device behaviors and are relevant models to apply model-checking techniques. The properties may be formalized as formulas of a formal logic interpreted over automata. In order to take advantage from well-known results about the properties preservation through parallel composition, we consider the ∀CTL* logic [9]. This logic is based on first order logic and offers temporal operators allowing to express properties holding for states and paths in the model. Nowadays, a large set of model checking tools supports ∀CTL* properties verification; moreover this logic is well suited to express safety properties.

The logic is interpreted over *Kripke structure* in order to express model checking algorithms and then satisfaction of a formula can be defined in a natural inductive way (see [9] for complete definitions). A mealy machine can be mapped to a Kripke structure (we do not detail this operation, it is fully described in [11]). We consider that a Mealy machine M satisfies a property ψ ($M \models \psi$), if and only if its associated Kripke structure ($\mathscr{K}(M)$) satisfies ψ.

Moreover, considering Mealy machines allows us to get an appealing preservation result. Indeed, we prove that if $M_1 \models \psi$ then $M_1 \| M_2 \models \psi$. Similarly to the work already done in [11], we prove that $\mathscr{K}(M_1 \| M_2)$ is an approximation of $\mathscr{K}(M_1)$. To show this result, we must define a surjective function between the state space of $\mathscr{K}(M_1 \| M_2)$ and the one of $\mathscr{K}(M_1)$ respecting transition relations. We consider a projection function $p : \mathscr{K}(M_1 \| M_2) \longmapsto \mathscr{K}(M_1)$ and prove that it has all the required properties to be an approximation. Then, we also define a translation function τ from formulas expressing properties in M_1 to formulas expressing properties in $M_1 \| M_2$ and finally we get the result: $M_1 \models \psi \Rightarrow M_1 \| M_2 \models \tau(\psi)$.

4. SAFE APPLICATIONS

Nowadays, IoT applications are ambient and self-adaptive. Thus middleware for the IoT must provide means to take into account "Thing" changes. Our proposal is to introduce in the design some

[1]Indeed, specific variables called registers encode states in implicit representation.

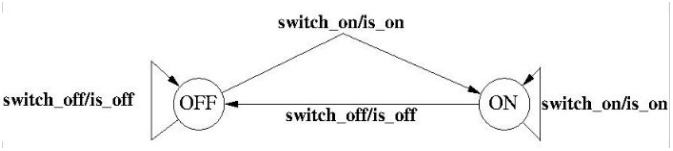

Figure 2: The heater behavior as an explicit Mealy machine: services are represented by output events : `is_on` and `is_off`. The heater has two switches input events `switch_on` and `switch_off` and two states ON , OFF.

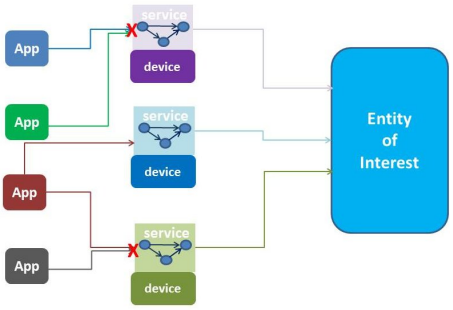

Figure 3: Concurrent accesses to services

constraint components whose purpose would be to correctly manage (according to the *Entity of Interest* and the application constraints) services offered by devices and used by applications. Constraint components are composed with synchronous models of device behaviors and a set of constraints describing how the devices output events are connected to the input events of the *Entity of Interest*.

4.1 Models of Services and Applications

First of all, we describe device behaviors as synchronous models. We introduce a component representing a device behavior. This component is designed as a Mealy machine, its outputs represent the device services. For instance, Figure 2 shows the explicit Mealy machine associated to both the AirConditioner and the heater devices introduced in section 2.

A "Thing" can concern numerous devices exposing services. These services are invoked by multiple applications, so multiple accesses problem to one of them can appears (Figure 3). In a previous work [11], a composition under constraints operation has been presented to address this problem. It consists in introducing a constraint com-

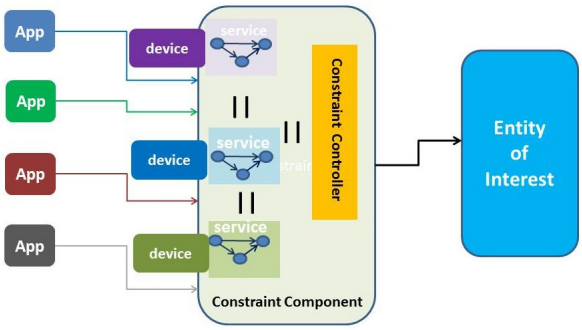

Figure 4: Our solution: a constraint component

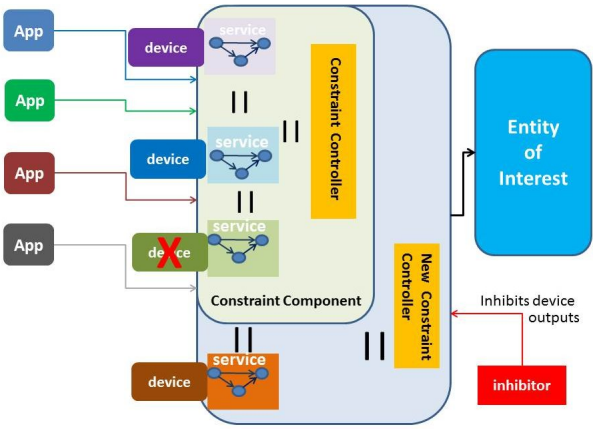

Figure 5: Device appearance and disappearance

ponent composed of the parallel composition of devices models on which a constraint function has been applied. This solution has several disadvantages since it is neither incremental nor adaptive to changes. Thus we propose another solution to face this problem. We introduce a constraint component managing (1) the combination of services and (2) the deployment of applications in a safe way. The constraint component is placed in parallel with Devices/Applications components (figure 4). It helps to make changes in the constraint combination, in the case of appearance or disappearance of a device, without the need of replacing the original constraint component. Hence, a complex operation (composition under constraints) is replaced by a traditional synchronous parallel composition. This component is build upon device models and a specific synchronous component (called *constraint controller*) that implements the set of constraints between device models and applications. Constraint controllers are combinatorial Mealy machines and they can easily be described as implicit Mealy machines with Boolean equations.

However, a problem we have to deal with is the communication between our synchronous constraint component, and both the applications and the entity of interest, which is by nature asynchronous. To resolve this issue, we create an asynchronous/synchronous transformer, composed of a synchronizer and a desynchronizer. The synchronizer is placed between the applications and the constraint component, it groups asynchronous data sent by the applications into synchronous instants and send them to the constraint component, according to grouping predifined policies. The desynchronizer is placed between the constraint component and the Entity Of Interest, it receives synchronous data from the constraint component and immerse them into asynchronous environment to send them to the Entity Of Interest[2]

To describe all the constraints related to n services connected to an *Entity of Interest* used by k applications, we introduce a language **DCL** (**D**escription **C**onstraint **L**anguage). Constraints descriptions using this language can be seen as annotations attached to applications and devices. Programs are automatically translated to synchronous component.

4.2 The DCL Language

4.2.1 DCL Overview

[2]The description of such a transformer is out of the scope of this article and will be detailed in an ongoing paper.

DCL is a language used to describe generic constraints. It allows the management of multiple accesses by handling a combination of all possible constraints, taking into consideration all existing devices and applications. It is devoted to specify a constraint component in a generic way only from the knowledge of device and application types. Input events must be defined corresponding to the services that can be invoked by applications. Then constraints are described as Boolean equations with a dedicated syntax allowing quantification for `or` and `and` operations. Constraints are twofold: (1) constraints related to service combination per device types, (2) constraints related to applications combination describing how the different applications use typed services. Thus, the only useful information is the knowledge of the types. DCL programs are compiled into implicit Mealy machines.

In a dynamic environment, applications and devices can appear and disappear. With DCL we can consider these two cases:

Appearance: Usually, applications are managed by an application scheduler instantiating all the necessary links between a new application and existing services. Services accesses are sorted by the scheduler. Nevertheless, applications and devices appearance are managed in the same way in our approach: a new constraint component is created by parallel composition from the original constraint component, a new device model (when a device is discovered) and a new constraint controller describing how the original constraint controller typed outputs have to be combined with the new ones (device model or application) (Figure 5). Moreover, according to the preservation result we have (see section 3.3), properties for the original constraint component are still true in the new one.

Disappearance: The scheduler is responsible for activating or blocking an application. If an application disappears, it links with the devices will also disappear and the devices stop their services. In the case of a device disappearance, only the equations in the constraint controller are inhibited. To this aim, specific entries (called inhibitors) are generated as constraint component inputs. There are as many inhibitor inputs as devices. As already mentioned, the constraint controller is an implicit Mealy machine, and its equations take into account inhibitors. Thus, when a device disappears, it is sufficient to set its inhibitor. The disappearance of a device can lead to the application linked to it to be blocked by the scheduler, if not used by any other device services (Figure 5).

4.2.2 DCL Issues

The goal of DCL is to allow a generic and compact representation of the connections between applications, devices and Entity of Interest, in order to be automatically derived in a particular context, where the number of devices is known and the relevant applications too. Indeed, the constraint component representation as Mealy machines is well known to be efficient. Moreover, when a lot of devices definitively disappear, we continue to handle useless connections and then we can choose to replace a constraint component by a new one, generated again from DCL annotation, that fits better the situation. Efficiency is the balance between weaving again or not. Regarding the scalability concern, the IoT handle a huge number of small entities and then according to our compositional approach, we have the abitity to minimize models.

On another hand, to design DCL annotations, the user must be aware of the *types* of the devices in the environment like those of the potential applications that may use device services. This notion of type is an abstract notion to represent knowledge, that could be refined. However, devices and applications of a given type have the same functionality and the user must know the functionality associated with each type.

4.2.3 DCL application to the Use Case

Due limited space, we cannot totally detail DCL. Instead, we just give a short description through the use case introduced in section 2. The constraint component generated for this example contains two instances of the Mealy machine described in Figure 2. The first instance represents the AirConditioner device behavior and the second one represents the Heater device behavior. The constraint controller is an implicit Mealy machine automatically generated from a DCL annotation. We briefly describe this annotation hereafter:

```
Constraint room_temperature :
OutputType: Controlled ;
ApplicationType: Cooling, Warming;
DeviceType: AirConditioner : n, Heater : m;
Interface: is_on, is_off : bool;
```

DCL language defines **constraints**. This declarative part describes the constraint's name. *Controlled* represents the general output type of the generated constraint component. It can further be combined with other DCL annotations. *Cooling* and *Warming* are application types. We consider two different device types (*AirConditioner* and *Heater*) and n is the number of *AirConditioner* devices type , m the number of *Heater* devices type . The two proposed services are *is_on* and *is_off*.

```
Constraints:
AirConditioner:is_on =
      Or i in [0..n-1] device[i].is_on and
      Forall j in [n..n+m-1] (device[j].is_off) ;
...

Application Output:
Cooling:is_on = AirConditioner:is_on ;
...

Output:
is_on = Cooling:is_on or Warming:is_on;
is_off = Cooling:is_off and Warming:is_off;
```

Then, constraints are defined. Firstly, constraints related to the combination of device model outputs are specified for each type. Then, for each application type, the outputs are defined as a combination of each outputs device types. Finally, the outputs *is_on* and *is_off* are the combination of application outputs. We express these constraints as Boolean expressions and we add quantifiers (**Forall** and **Exists**) along with n-aire Boolean operators (**Or** and **And**). All these constructs are bound and the generation is done for a fixed number of devices of each type. From this DCL annotation, we generate the constraint controller and we can construct the overall constraint component as a parallel composition with the explicit Mealy machine of *AirConditioner* and *Heater* devices. For instance, the implicit Mealy machine generated for $n = 1$ and $m = 1$ listens *air_conditioner-is_on*, *air_conditioner-is_off*, *heater-is_on*, *heater-is_off* input signals and emits *controlled_is_on*, *controlled_is_off* output signals.

We apply this technique in WCOMP [13] middleware, using a specific synchronous environment (CLEM).

5. PRACTICAL ISSUES

WCOMP [13] is a middleware to manage applications adaptation at runtime according to the available devices in the environment and the evolution of the physical context. At a first level, it allows to compose Web Services for Devices with dynamic components assemblies. At a second level, it uses an adaption mechanism that automatically produces new assemblies according to the devices availability over time. These assemblies sometimes share some

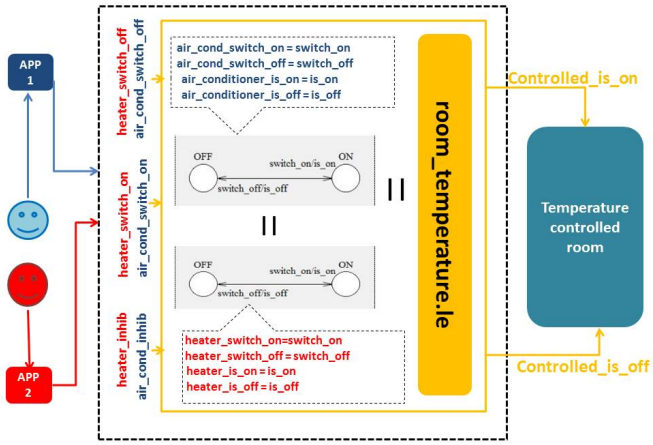

Figure 6: The room_temperature example. Inside the dashed line is the constraint component.

services. In this case the solution presented in this paper allows to generate intermediary components that guarantee safe accesses to shared Entities of Interest.

CLEM [3] solution is a set of tools devoted to design, the simulation, the verification and the generation of code for LE programs. LE is a synchronous language supporting a modular compilation. It offers synchronous languages' standard operators and more particularly a synchronous parallel operator. It also supports automata possibly designed with a dedicated graphical editor (GALAXY) (see Figure 2). Moreover implicit Mealy machine can be defined thanks to a Boolean equation format offered by LE. Thus, explicit and implicit Mealy machines are native constructs of LE. CLEM involves code generation to software and hardware targets. More particularly, an internal format is generated to support simulation and verification by model checking by invoking internal CLEM tools. CLEM verification tool applies model checking technique to prove that a signal is always or never emitted. CLEM also generates input code for the NuSMV model checker[4] (but using NuSMV requires to express properties as temporal logic formulas). Finally, CLEM automatically generates CSharp specific code for the new intermediaries component in WComp middleware.

To complete the use case introduced in section 2, we have generated the constraint component detailed in figure 6. To this aim, first we have generated a LE implicit Mealy machine from the DCL code in file *room_temperature.le*. The explicit Mealy machines describing both the behavior of AirConditioner and Heater devices are deduced from the Mealy machine described in Figure 2 in renaming the original inputs and outputs. This has been achieved calling the specific *run* operator of CLEM. The input signals of the global constraint component are: *air_cond_switch_on*, *air_cond_switch_off*, *heater_is_on*, *heater_is_off* which are listen respectively by the AirConditioner and Heater Mealy machines.
It also has *air_cond_inhib* and *heater_on_inhib* input signals which are listen by the constraint controller. These inhibition signals are generated by the DCL compiler. When a device is missing, its inhibitor is set to true. This solution is reversible and if the device

[3] http://www-sop.inria.fr/teams/pulsar/projects/Clem/
[4] http://nusmv.fbk.eu/

appears again, the inhibition can be canceled.

The second step is to perform some verification. Applying the CLEM model checker, we can prove on the implicit Mealy machine defined in *room_temperature.le* that (1) *air_conditioner_is_on* and *heater_is_off* implies *Controlled_is_on* and (2) *air_cond_inhib* and *heater_inhib* implies that *Controlled_is_on* is never emitted. Besides, if a property does not hold, counter examples can be simulated and help us to correct the designed annotations. Finally, when a successful verification stage is reached, the code embeding the constraint component in WCOMP is generated.

6. RELATED WORKS

Nowadays, IoT connects sensors, actuators and autonomous objects interacting each others. Current trends are the studies for large scale information systems that process a large amount of information collected on the field from numerous sensory devices. These systems can be used sometimes to pilot few actuators. Anyway, the main scientific challenge remains "Shared Sensing" or how to manage multiple applications collecting measures on a same set of connected sensors. Most of contributions then use different formalisms and propose various approaches to manage and optimize sensors accesses, often to avoid delays, bottlenecks and to limit communications bandwidth. For example in [3], the authors use an abstract model that can be coupled to timed-automata and code generation. In[12], the authors propose a timed automata based formalism to represent the functional part of design as well as the timing constraints in a unified way. In [1], the authors use also automata based models for the control and verification of their systems. These automata are related to ECA (Event-Condition-Action) rule semantics. In [8], Hugues and al generate Petri nets to model the Broker and use model checking techniques for Petri net models to verify qualitative properties. In this case, for example, model checking for Petri nets is difficult and exists only for particular classes (time Petri nets, simple Petri nets for instance). Some other approaches like *product line engineering* consider that systems are developed in families and differences between families members are explicitly expressed in terms of features. From this hypothesis, lots of works propose formal modeling and model checking techniques to describe and verify the combined behavior of a whole system family (see [4], for instance). In the same way, in [2] the authors present a compositional reasoning to verify middleware-based Software Architecture. They take advantage of the particular structure of applications due to their middleware-based approach to prove a property by decomposition into local properties verified on sub systems. These works do not allow the generic level we want to generate automatically validated components.

7. CONCLUSION

In this paper, we address the problem of services and applications composition safety in a middleware for IoT, which takes into account the context evolution through the available devices and observed physical environment. In our component based middleware, our solution generates specific components to manage accesses to shared services for devices. The goal of these components is to guarantee that the predefined constraints are always verified at runtime during the composition and adaptation. To provide safety, we apply formal methods offering exhaustive verification tools. Moreover, taking into account dynamic changes occurring in the environment (running applications and available devices), we use a runtime synchronous composition. Devices and applications appearances produce components thanks to a formal composition, whereas disappearances produces inhibitors for the missing objects in a previous composition. The main contribution of this work is the definition of a new language (DCL) (Description Constraint Language) that helps to manage the shared accesses to a set of devices and their *Entity of Interest*. (DCL) allows to describe generic constraints that must be verified on shared accesses. The whole approach and its associated tools rely on the synchronous paradigm, since it has a well-established formal foundation allowing automatic proof, and it interfaces nicely with most of the existing model-checkers. We can make the verification of our systems and thus prove and ensure composition safety. However, the (DCL) language must be improved to be really more user-friendly.

8. REFERENCES

[1] J. Cano, G. Delaval, E. Rutten, Y. Benazzouz, and L. Gurgen. Eca rules for iot environment: a case study in safe design. In *1st Workshop on Quality Assurance for Self-adaptive, Self-organising Systems (QA4SASO)*, Sept. 2014.

[2] M. Caporuscio, P. Inverardi, and P. Pelliccione. Compositional verification of middleware-based software architecture descriptions. In *Proc. of the 26th Int. Conf. on Software Engineering (ICSE'04)*, pages 221–230, Washington, DC, USA, 2004. IEEE Computer Society.

[3] C. Cecchinel, S. Mosser, and P. Collet. Software development support for shared sensing infrastructures: A generative and dynamic approach. In *Software Reuse for Dynamic Systems in the Cloud and Beyond*, number 8919 in LNCS, pages 221–236. Springer Int. Pub., Jan. 2015.

[4] A. Classen, P. Heymans, P.-Y. Schobbens, A. Legay, and J.-F. Raskin. Model checking lots of systems: efficient verification of temporal properties in software product lines. In *Proc. of the 32nd ACM/IEEE Int. Conf. on Software Engineering (ICSE'10)*, volume 1, pages 335–344, New York, NY, USA, 2010. ACM.

[5] A. K. Dey. Understanding and Using Context. *Personal Ubiquitous Comput.*, 5(1):4–7, Jan. 2001.

[6] D. Guinard, V. Trifa, and E. Wilde. A resource oriented architecture for the Web of Things. In *Proc. of the Int. Conf. on Internet of Things (IOT)*, pages 1–8. IEEE, Nov. 2010.

[7] S. Haller. The things in the internet of things. *Poster at the (IoT 2010). Tokyo, Japan, November*, 5:26, 2010.

[8] J.Hugues, L.Pautet, and F.Kordon. Refining middleware functions for verification purpose. In *Proc. of the Monterey Workshop (Monterey'03)*, pages 79–87, Chicago, IL, USA, September 2003.

[9] E. M. C. Jr., O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 2000.

[10] G. Mealy. A method to synthesizing sequential circuits. *Bell Systems Technical Journal*, pages 1045–1079, 1955.

[11] A. Ressouche, J.-Y. Tigli, and O. Carrillo. Toward validated composition in component-based adaptive middleware. In *Software Composition*, LNCS, pages 165–180. Springer Berlin Heidelberg, 2011.

[12] D. Socci, P. Poplavko, S. Bensalem, M. Bozga, and P. Bourgos. A timed-automata based middleware for time-critical multicore applications. In *Int. Symp. on Object/Component/Service-Oriented Real-Time Distributed Computing Workshops (ISORCW)*, pages 1–8, Apr. 2015.

[13] J.-Y. Tigli, S. Lavirotte, G. Rey, V. Hourdin, D. Cheung, E. Callegari, and M. Riveill. Wcomp middleware for ubiquitous computing: Aspects and composite event-based web services. In *Annals of Telecommunication*, volume 6. March-April 2009.