SALT: a Simple Application Logic description using Transducers for Internet of Things

Sylvain Cherrier, Yacine Ghamri-Doudane, Stephane Lohier, Gilles Roussel

SALT: a Simple Application Logic description using Transducers for Internet of Things

Sylvain Cherrier*, Yacine M. Ghamri-Doudane*§, Stéphane Lohier*, and Gilles Roussel*
* *Université Paris-Est , Laboratoire d'Informatique Gaspard-Monge (LIGM), 77454 Marne-la-Vallée Cedex 2*
§ *ENSIIE, 1 square de la résistance, 91025 Evry Cedex*
Email: [firstname.lastname]@univ-paris-est.fr

***Abstract*—As the Internet of Things (IoT) grows in interest from both research and industrial parts, the lack of standard solutions to quickly and easily build and install IoT applications becomes a topic of high interest. In this paper, we introduce a language called SALT (Simple Application Logic description using Transducers) that allows describing and deploying the distributed logic needed in order to fulfil a complete desired application. This language aims at giving extended functionalities by filling the gap between the logical capabilities offered by services orchestration and the closeness efficiency of services choreography. SALT is interpreted by a virtual machine running on devices that introduces an abstraction layer in order to simplify access to hardware capabilities. SALT implements several mechanisms to comply with organisation issues presented in the Services Oriented Computing realm dealing with Services interactions, adapted to the specific constraints of the IoT. This work details SALT's concepts, formalism and implementation.**

***Keywords*-Internet of Things; Architecture Design; Choreography; Distributed application logic.**

I. Introduction

Because everyday connected objects are promised to become the major user of public networks in a near future, Internet of Things is a main subject of interest for research and industry. Bringing Internet to Things adds a computational and digital dimension to users' physical environment. Interactions between Things, or with users, may have a great impact on individual relationships with the real world, thanks to the object connectivity.

The lack of standard and universal development platforms is one of the biggest barrier that impends the development of this new field. Users are facing a "tower of Babel" due to multiple involved technologies. Building an application using different objects leads to learn specific programming paradigms and/or languages and many other skills from various fields. Furthermore, applications are strongly linked to devices. Any hardware upgrade/failure may terminate the application, because replacement parts are probably incompatible. Currently, users are forced to limit themselves to a single manufacturer and/or hardware. These obvious and justified mistrusts greatly hamper the popularization of IoT.

Designing an Internet of Things app can be considered as building a collaboration of small independent elements. Each of these Things has a level of autonomy. SALT, proposed in this paper, is a programming language to take advantage of Object's autonomy. SALT's approach helps to design full distributed applications. In such an architecture, pieces of algorithms are spread all over the network. There is no central point. Each participant executes its own part of the work, acting or reacting to events or queries. Things have their own behaviour limited to their local perspective but coherent from their point of view. This organization is defined as *Choreography*.

On the contrary, any application under the control of a central point is built according to an architectural paradigm referred to as *Orchestration*. Data are centralized, computed and organized under the unique management of the central point. Each participant answers received requests from the central point and is not following a logic on its own.

To override constraints related to the strong coupling between hardware and software, we can consider setting a universal platform giving full hardware abstraction. Application creation, maintenance and evolution will be easier, quicker, scalable and loosely coupled to real elements at stake (see Fig 1). Our previous work, D-LITe [3], is a solution of this type. It is a framework running on Objects, introducing a hardware abstraction and discovering/deploying/executing services. Each object is seen as source of a logical service, and can subscribe to any other D-LITe object in order to build complex interactions.

Mixing D-LITe hardware abstraction and SALT Choreography design allows IoT applications to gain flexibility and versatility, acquiring standardized and universal formalization to express the programmer's needs while being understood by multiple devices.

This paper presents the SALT language design and implementation to express fully distributed applications over a hardware abstraction layer. The reminder is organized as follows: Section II presents the related work attached to Services Composition over the Internet, and then from the IoT point of view. Section III details the needs that SALT must be able to express, the language organisation and formalization. Section IV describes our implementation of SALT and our tools to validate the concept. Finally, concluding remarks and future research directions are given.

II. Background and related work

A. Services Oriented Architecture on Internet

In the Internet field, Service Oriented Architecture (SOA) becomes increasingly important as a solution because it hides the details of the different technologies by introducing abstraction. Abstraction provides the universality needed by programming languages to wrap all hardware/vendors specificities. The loosely coupling ability given by this approach helps to build applications easily.

[5] presents an analysis of Internet web services composition. The issue of checking the whole functionality arises because asynchronism appears in this composition. Authors checked that they obtain the same results using their proposed translation from a BPEL description (a normalized executable language used to build applications implying web services) to a more distributed and asynchronous collaboration. To move from Orchestration to Choreography, authors in [5] transform the centralized application BPEL description to a Finite State Machines (FST) combination. They propose to use a specific form of FST, called *guarded automata* (presented in [6]). Guarded automata introduce the notion of *predicate* which aims at adding more semantic in the exchanged messages. Authors transpose a central description into a distributed composition "*embed[s] the control flow [...] to the transitions of the guarded automata.*" [5]

Designing a Choreography needs to reproduce a majority of Orchestrated services interactions. In their paper dealing with *services choreography versus orchestration* [2], Barros et al identify 13 significant interaction patterns. If unicast communications can easily be described in both architectures, Orchestration introduces the possible use of enumeration functions and time-bound. These needs call for the introduction of management variables and timers on each participant when using a Choreographed architecture. Authors describe each case and propose a discussion about issues and design choices to achieve it. They don't propose any implementation of assessment.

The "Let's dance" language proposed in [7] is directly inspired by [2]. This choreographed language is a formal graphic language describing interactions between services according to patterns proposed in [2]. Emphasis is laid on the structural sides of services interaction. It consists in a description language used by Internet web services architects for the composition of pre-existing and not mutable web services. But it is more a tool to design a solution and to express the needs than an executable language. The same authors propose in [8] to fill the gap between the global view offered by their language and the local view (i.e the algorithm providing the service). Authors pointed out that "*some global models may not be translatable into a collection of local models such that the sum of the local models equals the original global model*". They propose an algorithm able to transform the "Let's Dance" global view

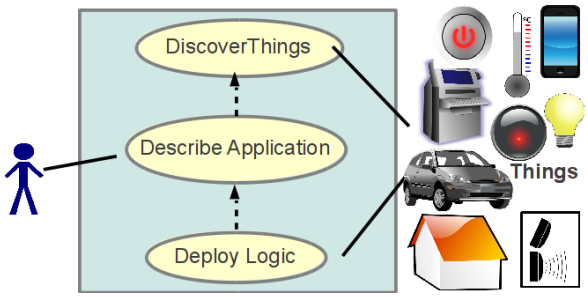

Figure 1. Use-Case: To build an Internet Of Things application, a user accesses in a universal way to his objects, discovers theirs capabilities, describes his needs, and deploys the code on each element

to real algorithms to be executed at local points. Therefore, when validated by their checking process, the combination of local actions gives expected results at the global level. This is a proof of concept and not a real executable language. Its potential enforcement to IoT is also questionable because of its heavy design which is not adapted to tiny devices.

Research work about Internet Services Composition is of great interest in the perspective of Internet of Things. In their survey [1], Atzori et al. present the IoT as a pervasive collaboration of objects "*that are able to interact with each other and cooperate with their neighbours to reach common goals*". By using widely spread and standardized Internet protocols, this paradigm gives Things the ability to communicate among each others and with Internet stakeholders (servers, applications, etc). But IoT applications also need to abstract the particularities of the hardware.

B. D-LITe

Our framework D-LITe (presented in [3]) implements a Choreographed design for Internet of Things devices. It also introduces a hardware abstraction to get rid of language/operating system/technical constraints. This framework is able to execute an algorithm expressed as a Finite State Transducer (FST). FST are automata with an additional output Alphabet, used in Natural Language Processing. A Finite State Transducer has a formal representation as a 6-tuple $T(Q, \Sigma, \Gamma, I, F, \delta)$. D-LITe defines the meaning of each element as follows:

- Q represents all *States* for a particular node,
- Σ are *Input Messages* handled by a particular node,
- Γ are *Output Messages* a particular node can send,
- I is the *Initial State* (only one in D-LITe),
- F stands for *Final States*,
- δ contains transitions (the distributed "*logic*").

ϵ element stands for empty. The main adaptation is the use of *Input Messages* and *Output Messages* in place of alphabets. These messages are used to exchange stimuli with other Objects, or to give access to hardware functions (see Fig 2). They abstract complex programming calls under

simple words provided by alphabet. The compact Transducer's description allows the transmission of the whole program through the network, changing dynamically the targeted Object's behaviour without any intervention on the device nor physical access. D-LITe offers only the hardware abstraction layer without describing the Transducers. SALT is the language describing such Transducers in order to give a syntax to IoT applications.

III. SALT: Simple Application Logic using Transducers

A. Identifying the Choreography needs

As we are interested in considering Objects as providers of web services, the SOA research community gives us several guidelines to orient our own solution. For IoT applications, we consider a rather top-down approach instead of composing already existing piece of software. SOA uses already existing Web Services and focuses on their combination. Usually, SOA applications are developed using a general approach, combining data provided by already-made and unalterable Web Services. In IoT application, the behaviour of each element is not as rigid, and so strictly pre-defined. It may depend on the context, on previous given reactions or on user's need. In IoT, many Objects are used by a reduced number of users (*one* user owned *many* Objects, while *one* Web Service is used by *many* people).

We can build new applications by describing needed services, creating and then deploying them. The result is a composition of distributed on-the-fly made services, dynamically created by the description of what is expected from Objects. That requires each participant to have sufficient capabilities to provide a result equivalent in terms of control structure to those offered by Orchestration. In addition, the more decisions are taken locally, on the spot, where events/stimuli are detected, the better it is for reactivity, energy consumption and reliability, especially if multi-hop wireless sensors networks are involved (see [4]).

Services Orchestration [2] may count how many answers a central application gets from a request, loops on a resource until a value is reached, or sets a period of time during which this algorithm waits for events. As a Choreographed design alternative, SALT must propose solutions to transfer all these constraints at the Object level.

To solve the possible lacks caused by moving the application logic from a central point to Things themselves, IoT Choreographies needs can be summarized as:

- being able to **count**, **compute** and **test** variable values. A choreographed Object not only provides atomic answers inferred from its environment, but rather more elaborated mechanisms, needed for its autonomy.
- being able to limit these more complex reactions over **time** in order to achieve greater independence and decision-making capacity.

Table I
Choreography issues, and SALT proposals

Describe an algorithm in each Thing	SALT can express a Transducer (an automaton embellished with an Output Alphabet) understood by Objects
Make Things cooperate	Transducer's Output alphabet is sent to other Things, becoming their Input alphabet. Things then react to each others, in cascade.
Algorithms' update (the application evolves), *or change* (new application)	The whole configuration description of an object's role through SALT has a very limited size, because it uses the framework installed on each Object. It can be totally reprogrammed with few network messages, quickly and without any physical access
Independence from hardware	SALT proposes an hardware abstraction. Hardware specificities are accessible through the Input or Output alphabet of the Transducer.
Need for computation and timers	Adding variables management and time control allows the object to gain in autonomy. SALT adds *predicate* as Input and Output alphabet to offer a greater level of reaction and expressibility.

[6] introduces the use of predicates in automata in Web Services Choreography. A predicate is a sentence containing a subject and information related to that subject. Unlike simple words of an alphabet, predicates have inner meanings. We use them to express mathematical operations and to test variables. Predicates in SALT can express the sentence "=(count,1)" which stands for *create a variable "count" to store the value 1*, or ">(i,7)" that means *if the variable i contains more than 7*.

As it extends use of predicates to *Input* and *Output* alphabets, SALT offers variable management and timers definition. This new use of predicates increases SALT expressibility. Each transition may describe logical expressions in a local scope. With SALT, decisions usually made at the top-level can be achieved at end-points (see Table I). Most of Orchestrated application control structures can therefore be distributed on a Choreography of Things.

It is important to note that our approach is *event-centric*. SALT is based on actions/reactions in order to represent an algorithm with a Transducer. If the hardware does not offer threshold setting for sensed values, SALT provides a way to manage these sensed values as logical variables. SALT expressibility offers to test these value and to react, moving from a *data-centric* approach to an *event-centric*'s one.

B. SALT Language Description

SALT makes a widespread use of *predicates* in *Input Messages* and *Output Messages* instead of usual Transducer's alphabets. In SALT, the programmer expresses his reasoning by the semantic he gives in messages content. A service on each Object receives the SALT description of the Transducer (see Fig 3). Then, the framework executes it and makes the link between this logical expression and actions offered by each specific hardware. In our vision of IoT applications, a Transducer always starts with one and only one *Initial State*. Then, it receives messages (considered as *Transducer*

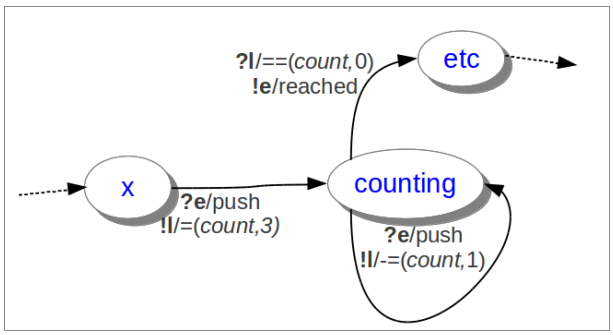

Figure 2. SALT Graphical representation, counting 4 "push" messages before sending "reached". For that purpose, a predicate creates a variable named *count* and another decrements it. 0 is detected by the last predicate.

Table II
SALT TEXTUAL REPRESENTATION OF EXAMPLE FIG 2

x ?**e**/push !**l**/=(count,3) *counting*
counting ?**e**/push !**l**/-=(count,1) *counting*
counting ?**l**/==(count,0) !**e**/reached *etc*

Input), checks if there is a *Transition* matching that input and the given state of the *Transducer*. If so, the framework obeys the transition and goes in the new state described in the *Transducer*. As a *Transducer*, it sends the *Output Message* as specified in the transition. Here, a transition has 4 items: The *Start State*, the *Received Message*, the *Next State*, and the *Output Message*. Finally, no *End state* is defined (IoT application has often no end). It is however possible to express such *End State* (i.e "alarm" in Fig. 5).

To do basic processing operations, to sense or actuate the real world, or to exchange with other Objects of an IoT application, we use *Input* and *Output* alphabets (See Fig 3). These messages are of 3 different kinds:

- *external messages* are used to organize the collaboration between Things. Content of exchanged messages will be analysed by transducers running on listening Objects (i.e. communication with other Objects)
- *hardware messages* make the link between the Thing's hardware and the programmer's logic. Messages are hardware specific and accessible via SALT's API.
  1) Sensing generates a SALT message for the Transducer *Input*,
  2) Things having actuating capabilities (e.g. switching on/off) react to Transducer's *Output*.
  3) Some sensing and actuating capabilities are virtual, such as timers. An *Output* hardware message starts the timer and defines the delay. At the end of this delay, the Transducer is alerted by an *Input* message that the timer has expired.
- *logical messages* are used to define, alter and test variables. The definition of a variable, its value or any operation are logical *Output* messages of the Transducer. On the contrary, testing the value of this variable is an *Input* message. The test becomes an event when its result is true.

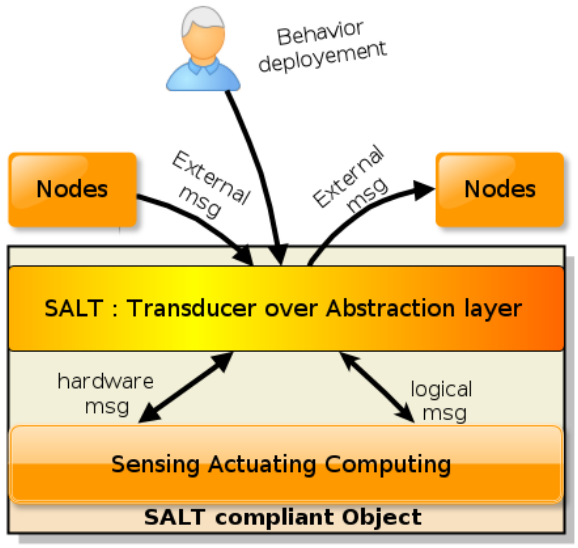

Figure 3. SALT architecture overview. After receiving user's description of the application logic, a SALT compliant Object reacts to external, hardware (sensing) or logical messages (testing variables). It may then send messages, alter variable, or actuate hardware.

Another way of understanding these different kinds of messages is to consider that external messages are intended for object's environment (useful to make others Things react) while hardware and logical messages are intended for object's internal use (sensing, actuating, or computing) as depicted in Fig 3.

C. SALT Language formalism

SALT uses Transducers and adds predicates to Input and Output Alphabet. Predicates are assertions or denials. It gives SALT more semantic on messages. SALT mixes the algorithmic usage of Transducers with the expressiveness of predicates to achieve a distributed logic.

The whole Transducer can be express as a set of Transitions. The formalism of a Transition description is given by Table III. As States are content free, a programmer can use any word that represents phases in his own reasoning. *Input* and *Output* Alphabets have a more constrained formalism, because of SALT's extended use of predicates. For each Transition, there is usually 1 Input message (Empty Input ϵ is a special case used to set variables or the Object's hardware). A transition can have 0 to many Output Messages:

- no Output message when the programmer wants to change state without altering anything,
- one or many messages depending on the actions to do.

A message is (**e**)xternal, (**l**)ogical, or (**h**)ardware, preceded by a question mark (if it's an input message) or an exclamation mark (in the case of an output message). It contains

Table III
SALT LANGUAGE: TRANSITION STRUCTURE

Transition Formalisation : **State Input Output ... State**	
State **xxxx**	A single word **xxxx**, giving semantic to programmer's reasoning. Has no effect on anything. It's only for naming the current stage in the Object's logic.
Input **?x/yyyy**	a **question mark**, follow by (x) (**e**, **l** or **h** for external, logical or hardware message). Predicate (**yyyy**) has different formalism. In external message, it is the expected message. In logical message, it is a predicate about a variable (a test). In hardware message, it is a pre-defined command using Object's sensing capabilities (described during Object's discovering phase).
Output **!x/yyyy**	a **exclamation mark**, follow by (x) (**e**, **l** or **h**). An external message is sent to all listeners, a logical predicate creates or alters a variable inside the Thing, while a hardware predicate actuates something. This action can be parametrised if needed, for example to set the hardware (i.e. *led(on)* or *led(off)*) or for processing operation (i.e *=(nb,5)* or *+=(count,2)*).
...	A transition may have multiple output messages (for hardware, listeners or variable management)
State **zzzz**	A simple word, describing the step reached by the application in programmer's mind.

Table IV
SALT LANGUAGE: PREDICATE TO EXPRESS VARIABLE MANAGEMENT

Input predicate (*testing variables*)	
?l/==(var,value)	Testing if "*var*" is equal to "*value*"... If true, the transition using **< > <= >=** and **!=** is executed
Output predicate (*setting or computing variables*)	
!l/=(var,value)	give "*value*" to "*var*" variable. The variable is created if unknown.
!l/+=(var,value)	adding "*value*" to "*var*" variable. *Value* can be another variable. The result is store in "*var*". 4 operations supported (**+=,-=, *=** and **/=**)

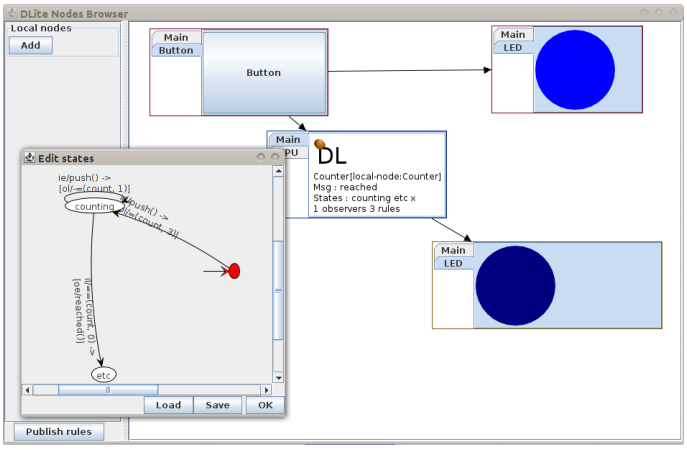

Figure 4. SALT GUI tool. On this example, a button (top left) controls a light (top right), but is also observed by a counter (counting 3 pushes) that triggers an indicator led (bottom right). The Transducer is displayed near its targeted Object. The "publish rules" button deploys the application.

a single word or something more semantic (a *predicate*: i.e. timer's duration, variable's value or test, external text to be displayed, etc).

- **External** messages contain a word of the alphabet (received for *input* messages, or sent for *output* messages).
- **Hardware** messages give the keyword corresponding to the event being managed. This complete set of managed keywords is provided during the Object's discovering phase. These words are either events sensed and/or generated by the hardware (in an *Input message*), or actions to trigger (in an *Output Message*). They can have zero to many parameters. Parameters are also presented during the discovery phase.
- **Logical** messages contain predicates for variable management and testing. They describe the operation and the variable to compute (see Table IV). As output predicate, SALT uses common expression (=, +=,-=,*= or /=) to set or modify the variable content (see Fig 2). As input predicate, tests use common expressions as well (==, *!*=,>,<,>= or <=).

SALT messages have the following roles:

- *External messages* are totally open and free texts, representing programmer's semantic (the interaction semantic between objects). They have no effect on hardware or logical functionalities.
- *Hardware messages* are very specific because they are responsible for interactions between SALT program's description and the hardware. Hardware messages are related to each different kind of devices, and provided during Object's discovery phase. They give the application the ability to sense/actuate, and hide the programming skills needed to use them.
- *Logical messages* are in charge of the definition of variables and the computation of theirs values. Creating/altering a given variable is always an *output* message, resulting of a transition. Testing the value is always a SALT *input* message. SALT manages the logical test result as an event: if true, the transition is triggered (see Fig 2 and 5)

IV. SALT IMPLEMENTATION AND EXAMPLES

We propose a tool to describe an application and experiment SALT[1]. This tool helps a programmer to declare his Objects, and to generate his complete SALT description. The resulting application can be executed (as the tool simulates Objects' behaviour). New applications involving the same Objects can be designed and executed on-the-fly, thanks to the expressiveness of SALT.

We also provide D-LITe implementation supporting SALT for real devices, in order to execute the Transducers. At the moment, we only support TelosB[2] sensors and Android phones (new devices will be supported in the future).

The first example presented in this paper (Fig 2) is actually running on Fig 4. The opened window shows the Transducer running on the selected Object "counter". Our tool gives the possibility to create Objects, and to

[1]http://www-igm.univ-mlv.fr/PASNet/SALT/

[2]A wireless sensor for research/experimentation http://www.memsic.com

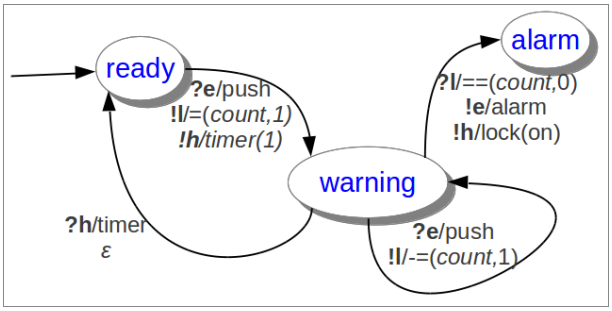

Figure 5. If the button is pressed twice ("push" messages are sent by inner hardware, because our object is a sensor) within one second, this alarm button gets locked (it's also an actuator) and sends an "alarm" message.

Table V
SALT TEXTUAL REPRESENTATION OF EXAMPLE FIG 5

```
initialState=ready
ready ?e/push !l/=(count,1) !h/time(1) warning
warning ?e/push !l/-=(count,1) warning
warning ?h/timer ready
warning ?l/==(count,0) !e/alarm !h/lock(on) alarm
```

describe Transducers and relationships between Objects. The programmer describes the transitions with the formalism presented in Tables IV, III, II and V and Fig 2 and 5. The tool stores an XML description of each Transducer and allows to run, change and re-deploy a simulated application. Transducers can also be sent on real hardware for real use.

A second example in Fig 5 shows a Transducer sending an alarm and locking a device if its button is pressed twice within a time frame of one second. It presents the main features of SALT:

- *logical* message: a variable (named "*count*") is defined and set to one, then decreased, and tested.
- *parametrized hardware* message: when reaching the end state, the device freezes (*lock*(**on**))
- *multiple* output (setting the *timer* to one second, and creating the "*count*" variable containing 1)

Starting from previous example, this new one can be quickly defined and deployed on the same Objects. Fig 5 shows the graphical form of our language while Table V gives the full program sent over the network. The whole program is 178 bytes long. A compressed version of the language for constrained network (such as Wireless Sensors and Actuators Network) is also proposed in order to further decrease the program size. In this version, state name are translated into numbers (states name are useless inside Objects, only useful in the programmer's mind) and alphabet strings become reference. This will thus lead to a 84 bytes program size for the example of Fig 5 when this one is deployed in constrained networks.

V. CONCLUSION

Creating quickly and easily universal applications fully editable and scalable is necessary for the wide adoption and the rise of the Internet of Things promising field, considering that the focus must be on easy creation of applications that fit to user's needs, deployed in a standardized way. Currently, interacting objects tend to promote a model in which there is no central point of reasoning, but rather a set of reflex actions spread all over the network. Because real applications may need more than a combination of simple reactions, our solution, the SALT language, offers rather an elaborated scale of actions in response to stimuli, more complex and richer, but still local. SALT is able to express the application's distributed logic, allowing each Thing to act and react from his own, thus meeting the expectations of all other elements in order to achieve the global application's goal. The hardware abstraction offered by SALT and the associated software platform give independence from the hardware manufacturer own API/OS/language, while providing mechanisms similar to usual algorithmic structures found in centralized architectures (variables, loops and tests). As for future works, our objective is to extend the language to introduce more external controls as well as coherence verifications, which are not considered in SALT for the moment.

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito. The internet of things: A survey. *Computer Networks*, 54(15):2787 – 2805, 2010.

[2] A. Barros, M. Dumas, and A. Ter Hofstede. Service interaction patterns. *Business Process Management*, pages 302–318, 2005.

[3] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. D-lite : Distributed logic for internet of things services. In *2011 IEEE International Conferences on Internet of Things, and Cyber, Physical and Social Computing*, pages 16–24. IEEE, 2011.

[4] S. Cherrier, Y. Ghamri-Doudane, S. Lohier, and G. Roussel. Services Collaboration in Wireless Sensor and Actuator Networks: Orchestration versus Choreography. In *17th IEEE Symposium on Computers and Communications (ISCC'12)*, page 8 pp, Cappadocia, Turquie, July 2012.

[5] X. Fu, T. Bultan, and J. Su. Analysis of interacting bpel web services. In *Proceedings of the 13th international conference on World Wide Web*, pages 621–630. ACM, 2004.

[6] X. Fu, T. Bultan, and J. Su. Model checking interactions of composite web services. *UCSB Computer Science Department Technical Report (2004-05)*, 2004.

[7] J. Zaha, A. Barros, M. Dumas, and A. Ter Hofstede. Let's dance: A language for service behavior modeling. *On the Move to Meaningful Internet Systems 2006: CoopIS, DOA, GADA, and ODBASE*, pages 145–162, 2006.

[8] J. Zaha, M. Dumas, A. Ter Hofstede, A. Barros, and G. Decker. Service interaction modeling: Bridging global and local views. In *Enterprise Distributed Object Computing Conference, 2006. EDOC'06. 10th IEEE International*, pages 45–55. IEEE, 2006.