Rigorous Design and Deployment of IoT Applications

Ajay Krishna*, Michel Le Pallec†, Radu Mateescu*, Ludovic Noirie† and Gwen Salaün‡
*Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG
38000 Grenoble, France
†Nokia Bell Labs
91620 Nozay, France
‡Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG
38000 Grenoble, France

***Abstract*—Internet connected devices are becoming increasingly common in consumer homes. These devices combined with software entities are used to build Internet of Things (IoT) applications. As democratization of IoT takes shape, developing reliable IoT applications remains a challenge for consumers because these applications exhibit characteristics such as distribution, dynamicity, and heterogeneity, which make their design, development and maintenance difficult. In this paper, we use formal methods to ensure correct composition of objects and propose a reliable deployment mechanism in the context of an IoT application. Objects are modelled using an interface description model integrating a behavioural specification of the object functionality. We provide formal validation techniques for verifying that the composition is correct. A deployment plan is generated for automating the instantiation of all objects involved in a valid composition. All the proposals have been implemented as a prototype tool and experiments were carried out for evaluating the tool performance and usability.**

***Index Terms*—IoT, behavioural modelling, composition, formal verification, deployment**

I. Introduction

The Internet of Things (IoT) aims at sensing and altering our surrounding environment through connected devices to improve everyday life. Nowadays, the IoT specific hardware, especially to collect the information, are already in place, but there is still room for improvement regarding support on building of IoT applications, which are supposed to empower interconnected objects in order to build powerful and added-value services. Designing, developing, and maintaining IoT applications are difficult tasks, because such applications are highly distributed, data-centric, dynamic, and heterogeneous. First, IoT applications induce a high level of concurrency, distribution, and collaboration. Designing concurrent applications is a tedious and error-prone task, in particular because software is more and more complex. Complexity is intrinsic to IoT scenarios where we can imagine in a short-term future thousands of interconnected objects [1]. Further, modern systems are no longer designed and implemented once-for-all, they can evolve over time. Last, there are multiple standards for communication protocols and technologies. Therefore, IoT applications have to deal with heterogeneous IoT hardware and communication layers.

The focus in this paper is on IoT object composition in order to build an application relying on objects and to provide a composite service of value for smart home users. Composition of objects is however a difficult task for several reasons. First of all, composition relies on models of the devices to be composed. Several levels of expressiveness can be considered [2] (signature, behaviour, semantics, quality of service) and each facet brings different issues from a composition perspective. Once a model of objects is properly defined, one can design a composition by specifying connections or bindings among the involved objects. Yet, building such a composition is error-prone because the objects might not have been connected correctly by the developer. Depending on the object description model, several kinds of mismatch can arise. So there is a need for analysis techniques in order to validate the composition and ensure that before the composition is deployed it works correctly. Lastly, tool support should be available in order to help the end-user during the composition and deployment tasks and make them as automated as possible.

In this paper, we propose various techniques to support the composition and deployment of objects in the context of an IoT application (Figure 1). The description model we consider for objects integrates the signature and behaviour levels. Our approach aims at supporting the IoT object composition via a user-friendly web application which helps the end-user to choose and connect the objects. The composition solution also provides validation techniques for verifying its correctness. Finally, a deployment plan is generated, which guides the automated instantiation of all objects involved in the composition in a certain order.

More precisely, the model consists of a set of signatures (interfaces) and a behavioural model (Labelled Transition System) giving the order in which these interfaces must be called. Given the models of a set of objects, the end-user can define how these objects are supposed to interact with one another. Once these bindings are defined, we propose automated analysis techniques in order to check whether the composition under construction satisfies some properties of interest, called compatibility notions in the rest of this paper. As an example, we can check that every binding can be executed, meaning that a pair of objects which are bound

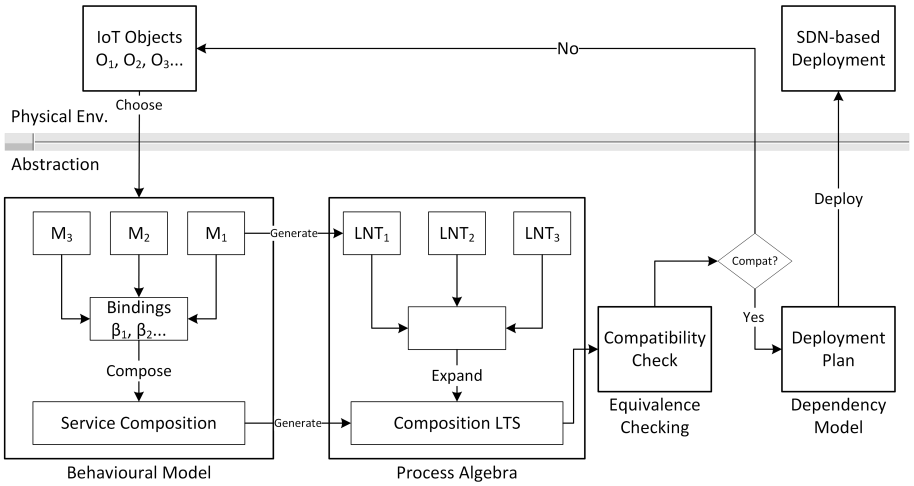

Figure 1. Verified IoT composition and deployment.

through given application interfaces can execute its complete behaviour, i.e., none of the objects lose any of its capabilities. Those checks are achieved using process algebra [3] encodings and equivalence checking techniques [4]. If the execution is not satisfied, this results in an erroneous composition. When the composition is valid, a deployment plan is generated and can be run to effectively interface and execute these objects. The plan is applied in a certain order to respect the dependency graph inherent to the composition model. The underlying idea is that if an object O_1 depends on an object O_2, we start first O_2 and then O_1. We implemented a web application for simplifying the use of our approach and we applied it successfully to several real-world applications.

To summarize, these are the main contributions of this paper: i) a formal model for objects and their composition. ii) compatibility notion and encoding into process algebra for automating these checks. iii) deployment plan generation and execution for running composition of objects. iv) tool support for helping the end-user during the composition and deployment steps. v) application to real-world use-cases in smart homes for validation purposes.

The organization of this paper is as follows. Section II introduces the model of objects and composition we rely on in this paper. Section III defines compatibility notions and how we check them in this work. Section IV focuses on deployment plan and its execution. In Section V, we describe our tool support and experiments. Section VI overviews related work and Section VII concludes the paper.

II. Models

In this section, we present a model for IoT objects, and use it to define a composite IoT service. This model provides a formal basis for reasoning about the behaviour of IoT objects, by allowing one to describe the behaviour of IoT objects in an abstract way, independent from implementation details, and to represent only the features necessary for correct object composition and deployment.

A. IoT Object

The behaviour of an IoT object is modelled as a Labelled Transition System (LTS), i.e., a state-transition graph representing the states of the object and its interactions with the environment.

Definition 1 (Labelled Transition System): An LTS is a tuple $\langle S, A, T, s_0 \rangle$, where S is a set of states, A is a set of actions (labels) associated to transitions, $T \subseteq S \times A \times S$ is the transition relation, and $s_0 \in S$ is the initial state. A transition $(s_1, a, s_2) \in T$ (also noted $s_1 \xrightarrow{a} s_2$) indicates that the system can move from state s_1 to state s_2 by performing action a. The set A includes the internal action τ, which denotes unobservable behaviour.

An IoT object (also referred to as object or device in the sequel) contains physical, network, and application interfaces, which are used to physically connect the object (to a network or to its environment), to provide network access to the object, and to define its behaviour, respectively. When the object is connected to a network, we refer to it as a Connected Object (CO). Given the application context, we focus here on application interfaces. However, if we are to extend the model with information pertaining to physical interfaces, a more detailed model like the one in [5] can be integrated with the proposed model.

Definition 2 (IoT Object): An IoT object $O = \langle I_{in}, I_{out}, LTS \rangle$ consists of a set of input (application) interfaces I_{in}, a set of output (application) interfaces I_{out}, and an LTS $\langle S, A, T, s_0 \rangle$ describing the object behaviour, where $A \subseteq I_{in} \cup I_{out} \cup \{\tau\}$.

We assume in our approach that each object comes with an object model. These models can be defined by an expert who

has the knowledge of the device (i.e., device manufacturers can provide the model along with the datasheet) or they can be built by learning the behaviour of these devices [6].

B. Object Composition

Composition aims at connecting various objects through their application interfaces, resulting in a composite service. The objects involved in a composition interact with each other through bindings in a binary synchronous manner.

Definition 3 (Binding): Given two objects O_1 and O_2, a binding is an ordered pair $\beta = (i_{out}^{O_1}, i_{in}^{O_2})$ specifying the connection between the output interface $i_{out}^{O_1}$ of O_1 and the input interface $i_{in}^{O_2}$ of O_2. For a binding $\beta = (i_{out}^{O_1}, i_{in}^{O_2})$, we note $in(\beta) = i_{in}^{O_2}$ and $out(\beta) = i_{out}^{O_1}$.

From the composition perspective, we distinguish between *strong* and *weak* bindings. In our setting, *strong* bindings are functionally mandatory connections, without which a correct service cannot be achieved by the composition. On the other hand, *weak* bindings are optional for a correct functioning, but provide additional features to the composite service.

Some auxiliary notation is needed before defining composition. An action renaming $\rho = [b_1/a_1, ..., b_n/a_n]$ is a partial function that maps each action a_i to another action b_i for $1 \leq i \leq n$, i.e., $\rho(a)$ is equal to b_i if $a = a_i$ and is equal to a otherwise. The renaming induced by a set of bindings $B = \{\beta_1, ..., \beta_m\}$, which maps the input and output interfaces of each binding β_i to the name β_i, is defined as $\rho_B = [\beta_1/in(\beta_1), \beta_1/out(\beta_1), ..., \beta_m/in(\beta_m), \beta_m/out(\beta_m)]$.
For simplicity, we denote by β the name of a binding β. The notations *in* and *out* are extended to sets of bindings: $in(B) = \{in(\beta_1), ..., in(\beta_m)\}$ and $out(B) = \{out(\beta_1), ..., out(\beta_m)\}$. The renaming of an LTS by ρ is defined as $ren_\rho(\langle S, A, T, s_0\rangle) = \langle S, \{\rho(a) \mid a \in A\}, \{s \xrightarrow{\rho(a)} s' \mid s \xrightarrow{a} s' \in T\}, s_0\rangle$. The parallel composition of $LTS_1 = \langle S_1, A_1, T_1, s_{01}\rangle$ and $LTS_2 = \langle S_2, A_2, T_2, s_{02}\rangle$ with synchronization on a set of actions $A \subseteq A_1 \cup A_2$ is written as $LTS_1 \otimes_A LTS_2$. This parallel composition is itself an LTS

$$\langle S = S_1 \times S_2, A' = A_1 \cup A_2, \\ T \subseteq S \times A' \times S, s_0 = (s_{01}, s_{02})\rangle,$$

where the transitions in T are defined as follows:
(i) $(s_1, s_2) \xrightarrow{a} (s_1', s_2)$ if $a \notin A$ and $s_1 \xrightarrow{a} s_1' \in T_1$;
(ii) $(s_1, s_2) \xrightarrow{a} (s_1, s_2')$ if $a \notin A$ and $s_2 \xrightarrow{a} s_2' \in T_2$;
(iii) $(s_1, s_2) \xrightarrow{a} (s_1', s_2')$ if $a \in A$ and $s_1 \xrightarrow{a} s_1' \in T_1$ and $s_2 \xrightarrow{a} s_2' \in T_2$.

This LTS, called the *synchronous product* between LTS_1 and LTS_2, can be constructed starting at the initial state $s_0 = (s_{01}, s_{02})$ and exploring the transitions according to the three rules above.

Definition 4 (Composition): A composition is a tuple $C = \langle B, \Sigma, I_{in}^U, I_{out}^U, LTS, W\rangle$, where $B = \{\beta_1, ..., \beta_m\}$ is a set of bindings, $\Sigma = \{O_1, ..., O_n\}$ is a set of objects, $I_{in}^U = (I_{in1} \cup ... \cup I_{inn}) \setminus in(B)$ is the set of unbound input interfaces, $I_{out}^U = (I_{out1} \cup ... \cup I_{outn}) \setminus out(B)$ is the set of unbound output interfaces, $LTS = ren_{\rho_B}(LTS_1) \otimes_B ... \otimes_B ren_{\rho_B}(LTS_n)$ is the LTS of the composition, and $W \subseteq B$ is the set of weak bindings.

The composition takes into account the future extensions to the service. It is an open-ended system, wherein every interface in an object O_i is not expected to be bound to another object O_j. The unbound (without active binding) input and output interfaces I_{in}^U and I_{out}^U can be used to bind other objects when the service is extended. This is typically the case in IoT service scenarios, where all features of an IoT device are not used in a given scenario. In certain cases, unbound input interfaces can interact with the real-world environment. Environment can be a human, a physical object or an event in the real world. We can define these interactions by modelling the behaviour of the environment as a specific object and establish bindings between the environment model and other objects in the composition. In other words, these environment bindings close the open-ended composition. Further, a composition $C = \langle B, \Sigma, I_{in}^U, I_{out}^U, LTS, W\rangle$ can be seen itself as an IoT object $\tilde{C} = \langle I_{in}^U, I_{out}^U, hide_B(LTS)\rangle$, where $hide_B(LTS) = ren_{[\tau/\beta_1,...,\tau/\beta_m]}(LTS)$ is the hiding of actions B in the LTS. This enables a hierarchical composition of services from individual devices and existing composite services.

Example 1: Consider an IoT composition scenario involving three objects, namely a smartphone app, a media device and an external speaker, illustrated in Figure 2 (input and output interfaces are represented by donuts and triangles, respectively). The smartphone app can connect to the media device and view the videos available on the phone screen. The media device can also be connected to an external speaker. These objects are modelled as follows:

$$\begin{aligned} Smartphone{=}&\langle\{\mathsf{P_VIDEO}\}, \{\mathsf{P_ON}\}, LTS_{Smartphone}\rangle \\ MediaDevice{=}&\langle\{\mathsf{M_ON}, \mathsf{M_AUX}\}, \{\mathsf{M_VIDEO}, \mathsf{M_AUDIO}\}, \\ &LTS_{Media}\rangle \\ Speaker{=}&\langle\{\mathsf{S_AUDIO}\}, \emptyset, LTS_{Speaker}\rangle \end{aligned}$$

The connections between objects are modelled by the bindings $\mathsf{ON} = (\mathsf{P_ON}, \mathsf{M_ON})$, $\mathsf{VIDEO} = (\mathsf{M_VIDEO}, \mathsf{P_VIDEO})$, and $\mathsf{AUDIO} = (\mathsf{M_AUDIO}, \mathsf{S_AUDIO})$. The composition is defined as

$$\begin{aligned} C = \langle &\{\mathsf{ON}, \mathsf{VIDEO}, \mathsf{AUDIO}\}, \\ &\{Smartphone, MediaDevice, Speaker\}, \\ &\{\mathsf{M_AUX}\}, \emptyset, LTS, \{\textit{VIDEO}\}\rangle \end{aligned}$$

The LTSs of individual objects are shown in Figure 2 (for the sake of simplicity, we have avoided loops in the behaviour), and the LTS of the overall composition is shown in Figure 3 (left). Notice that even though the media device provides an interface to add an auxiliary input, the composition does not support the operation. So, the interface $\mathsf{M_AUX}$ is unbound in the given composition, i.e., $I_{in}^U = \{\mathsf{M_AUX}\}$.

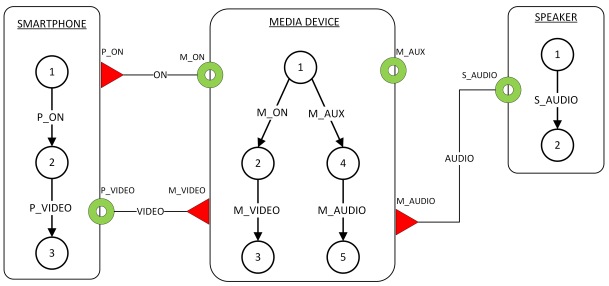

Figure 2. An IoT composition using three connected objects.

III. Compatibility

In Section II, we modelled a composition formally, here we investigate how to ensure its correctness from an interaction perspective. We first propose a notion of compatibility assessing that the bindings present in a composition do not alter the behaviour of the objects in the given setting. Then, we show how this compatibility is checked via an encoding into process algebra and an analysis based on equivalence checking.

A. Compatibility Notion

Intuitively, a composition involving a set of objects and a set of bindings is correct if all bindings can effectively be executed and if all reachable actions unbound in the composition do not prevent the bindings to be executed. We define this notion of compatibility using classic concurrency concepts introduced in Section II: parallel composition of LTSs with synchronization on a set of actions A ($LTS_1 \otimes_A LTS_2$), hiding of a set of actions A in an LTS ($hide_A(LTS)$), and branching equivalence [7] of two LTSs ($\equiv_{br}$).

Definition 5 (Compatibility): Let $C = \langle B, \Sigma, I^U_{in}, I^U_{out}, LTS, W \rangle$ be a composition involving a set of objects $\Sigma = \{O_1, ..., O_n\}$ with $LTS_i = \langle S_i, A_i, T_i, s_{0i} \rangle$ for $1 \leq i \leq n$. The composition C is correct with respect to compatibility iff:

$$hide_{I^U_{in} \cup I^U_{out}}(LTS) \quad \equiv_{br} \quad LTS \otimes_A chaos_B$$

where $A = A_1 \cup ... \cup A_n$ and $chaos_B = \langle \{s\}, B, \{s \xrightarrow{\beta} s \mid \beta \in B\}, s \rangle$ is the LTS consisting of a single state and having, for each binding, one self-loop transition labelled by the identifier of that binding.

Definition 5 expresses that, when put in an environment that blocks the execution of the unbound interfaces (right hand side of the equivalence), the composition still behaves as defined by the LTSs of the objects and the bindings, with the unbound interfaces made unobservable (left hand side of the equivalence). Similar compatibility notions between behavioural interfaces were defined in the services domain [8].

Example 2: Consider again the composition described in Example 1 and Figure 2, which involves a phone, a media device, and an external speaker. Although the bindings are valid from an interface point of view, the overall composition is not able to provide the desired service. Indeed, as defined by its LTS, the media device requires an auxiliary input on interface M_AUX for an audio M_AUDIO to be sent out. Since interface M_AUX is not bound in the composition (i.e., $I^U_{in} = \{\text{M_AUX}\}$), the environment may prevent it to occur, which in turn will make the action M_AUDIO (bound by AUDIO) unreachable. This is captured by the compatibility check, as illustrated in Figure 3. In the broad sense, incompatible object compositions result in missing features in an IoT application.

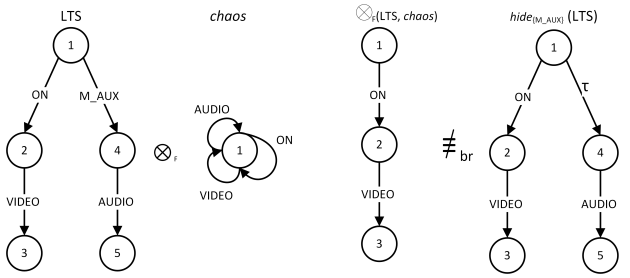

Figure 3. LTSs underlying the compatibility check for Example 1.

According to Definition 4, the LTS of the composition (leftmost in Fig. 3) synchronises on the bindings ON, AUDIO, and VIDEO, whereas M_AUX would evolve as an independent transition. Composing the LTS in parallel with $chaos_{\{\text{ON,AUDIO,VIDEO}\}}$ with synchronization on all actions $\{\text{ON}, \text{AUDIO}, \text{VIDEO}, \text{M_AUX}\}$ would eliminate the right branch of the LTS, resulting in the sequential system shown in Figure 3. This is clearly not branching equivalent to $hide_{\{\text{M_AUX}\}}(LTS)$ (rightmost in Fig. 3), because the latter system can silently perform the τ-step, followed by action AUDIO.

This notion of compatibility ensures that the LTSs $hide_{I^U_{in} \cup I^U_{out}}(LTS)$ and $LTS \otimes_A chaos_B$ have the same observational behaviour. However, it does not guarantee that this behaviour indeed corresponds to the desired service implemented by the composition. The intended behaviour can be ensured by verifying relevant functional properties (e.g., absence of deadlocks and livelocks, safety, liveness, fairness, etc.) on either LTS using, e.g., temporal logic and model checking techniques.

B. Analysis

The compatibility check is achieved using an encoding into process algebra. This allows us to rely on existing verification tools for process algebra and avoids developing new *ad-hoc* tools for automating this check. More precisely, we use here the LNT process algebraic language [3], [9] and the CADP toolbox [4], which accepts LNT as main specification language and supports different kinds of verification, such as model and equivalence checking.

LNT allows the definition of data types, functions, and processes. LNT processes are built from actions, choices (select), parallel composition (par), action hiding (hide), looping behaviours (loop), and sequential composition (;).

The communication between parallel processes takes place by rendezvous on a set of synchronized actions specified in the parallel composition.

```
module mediadevice is
 process
     mediadevice_idle[on, video, aux, audio:any]
     is
  select
   on; mediadevice_video[on, video, aux, audio]
  []
   aux; mediadevice_audio[on, video, aux, audio]
  end select
 end process
 process
     mediadevice_video[on, video, aux, audio:any]
     is
   video;
   mediadevice_exit[on, video, aux, audio]
 end process
 process
     mediadevice_audio[on, video, aux, audio:any]
     is
   audio;
   mediadevice_exit[on, video, aux, audio]
 end process
 process
     mediadevice_exit[on, video, aux, audio:any]
     is
  stop
 end process
end module
```

Listing 1. LNT processes specifying the *Media Device* LTS.

To carry out the compatibility check, we need to generate several LTSs as specified in Definition 5. We first encode each object LTS in LNT by associating an LNT process to each state of the LTS. The process body consists of a choice (select) specifying all transitions going out of that state, each one being followed by a call to the process associated to its target state, as shown in Listing 1 for the *Media* LTS. Then, the overall LTS of the composition is obtained by putting all object processes in parallel and synchronizing them on all binding names, as illustrated in Listing 2. Next, $chaos_B$ is encoded as an LNT process consisting of a cycle (loop) that chooses at each step (select) one of the binding names in B. Finally, the two processes $hide_{I^U_{in} \cup I^U_{out}}(LTS)$ and $LTS \otimes_A chaos_B$ are encoded in LNT using the hiding operator (hide) and the parallel composition with synchronization on all actions, respectively.

```
module prodall (phone, mediadevice, speaker) is
process prod [on, video, aux, audio:any] is
  par
    on, video -> phone_idle[on, video]
  ||
    on, video, audio ->
        mediadevice_idle[on, video, aux, audio]
  ||
    audio -> speaker_idle[audio]
  end par
end process
end module
```

Listing 2. LNT process specifying the composition of objects.

We generate the LTSs $hide_{I^U_{in} \cup I^U_{out}}(LTS)$ and $LTS \otimes_A chaos_B$ by invoking the CADP compilers on the corresponding LNT processes, and we apply the CADP equivalence checker to compare these LTSs modulo branching equivalence. The LTS generation and comparison steps are automated using SVL scripts [10].

IV. Deployment

This section describes the deployment of applications. The modelling and verification techniques proposed in the previous sections ensure that the object composition provides a correct IoT service. Further, we propose to generate a deployment plan based on the bindings. The plan aims at orchestrating the IoT service deployment which involves the control and configuration of objects and network layer.

By binding objects in a composition, we create dependencies among the objects. These dependencies must be taken into account during deployment in order to fulfil the service level agreements such as order of execution, availability, etc. A composition can be represented naturally as a directed graph, whose nodes are objects and whose directed edges are the bindings between objects. Using this directed graph, the dependency chain can be deduced using inverse topological sorting [11]. The limitation of this technique is that it works only on directed acyclic graphs (DAGs). If the graph of the composition contains cycles, we need to break these cycles (feedback arc set) and thus transform the graph into a DAG. It is here that the notion of *weak* and *strong* bindings comes into play. The edges corresponding to weak bindings that are present on cycles are discarded by the sorting algorithm, as they do not impede the primary service goal. At this point, we do not handle bindings involving strong cyclic dependencies.

Definition 6 (Deployment Plan): A deployment plan P consists of a sequence of steps involving three operations - [ADD, BIND, START] on an IoT object O.

ADD operation takes object O as input. *BIND* takes two different objects O_1 and O_2 and establishes a binding β in the composition. Similarly, *START* operation takes the object O to be started as input. More specifically, *ADD* refers to the provisioning of required application interfaces (e.g., 'motion detection' for a camera). *BIND* refers to the configuration of the network layer to allow the communication between application interfaces. *START* enables application interfaces of objects. The deployment plan dictates the order in which these operations are applied.

Example 3: As an illustration, consider a use case involving three devices: a baby monitor (dlink50201), a smartphone (phoneapp), and a TV (Chromecast). In this scenario, the baby monitor and the TV are co-located in a smart room whereas the smartphone is located in a remote environment. Assume that the TV (or any device screen) is installed in the baby's room. The user would like to turn on the screen and play baby's

favourite show on the screen, when he/she wakes up. It will keep the baby engaged for a while until he/she is attended by his/her parents or caretaker. The use case composition is shown in Figure 4. In order to restrict the unbound behaviour, we have defined the environment in the use case as shown in far left of Figure 4. A deployment plan as shown in Listing 3 is generated for this composition.

In this use case, the composition is without cycles. So, the dependency chain is pretty clear: the smartphone interfaces S_TVON and S_TVOFF depend on the TV (T_TVON, T_TVOFF) and the baby monitor B_ALERT depends on S_ALERT interface of the smartphone.

```
{"objects":[dlink50201, phoneapp, chromecast],
"bindings": [{
"id":"alert",
"source":"dlink50201_alert",
"target":"phoneapp_alert",
"type":"strong" },
{"id":"tvon",
"source":"phoneapp_switchon",
"target":"chromecast_switchon",
"type":"weak" },
{"id":"tvoff",
"source":"phoneapp_switchoff",
"target":"chromecast_switchoff",
"type":"weak" } ],
"plan": [{ "step": "ADD",
"element": "chromecast" },
{ "step":"ADD", "element": "phoneapp" },
{ "step":"BIND",
"element":"tvon" },
{ "step":"BIND",
"element":"tvoff" },
{ "step":"START",
"element":"chromecast" },
{ "step":"START",
"element":"phoneapp" },
{ "step":"ADD",
"element":"dlink50201" },
{ "step":"BIND",
"element":"alert" },
{ "step":"START",
"element":"dlink50201" }]}
```

Listing 3. Deployment plan in JSON.

The devices are deployed in the following order: first, the TV is deployed, then the smartphone, and finally the baby-monitor is deployed thereby preventing send-receive mismatch at the interface level. Further, in Section V, we detail how the deployment plan is leveraged by an existing platform for network level deployment.

V. Tool Support and Experiments

In this section, we describe our tool implementing the approach. The implementation is available online on Github[1]. Further in the section, we validate the approach using experiments involving composition and deployment of real-world IoT use cases.

[1]https://ajaykrishna.github.io/iotcomposer/

A. IoT Service Assistant

The target users of the tool may not have the required skills to generate correct compositions and deployments. So, the tool aims at guiding the user towards deployment by hiding the underlying complexities. The user-base that we are targeting is home users, who have the basic understanding of IoT devices, but they do not want to get into API-level programming to get things working. A short training session may be required for the users to familiarise themselves with the tool features.

The tool is hosted as a web application on an Apache Tomcat server with an intuitive user interface based on jQuery and Semantic UI. The data format used is JSON and the integration with other services is done via REST API calls. The assistant performs these major functions: i) initial recommendation of devices for composition, ii) object composition by allowing users to compose IoT services through binding devices spread over different smart environments, iii) compatibility check to ensure correctness of the designed IoT service at the application level, iv) automated deployment of the validated composition.

Recommendation: First step in the tool workflow is recommendation. In [12], the authors proposed an automated recommendation of IoT services based on the physical interaction between objects and their environments. We build upon their work and use the recommendation service to get a set of objects available for a possible composition. From the set of objects, users can choose the subset of objects to be involved in the composition. Next, the tool loads the corresponding object models (described in Section II), which are specified in JSON. Users can view the chosen objects on the web interface, along with their application interfaces.

Composition: In the composition step, users can graphically define a set of bindings between application interfaces of objects to generate a composition. Then, the defined bindings are stored in a JSON file, which is used to generate a deployment plan as described in Section IV. The web interface provides an option to graphically view the dependencies among devices in the form of a directed graph.

Compatibility: Next, the user is provided with an option to check the compatibility of the proposed composition. The tool connects to the backend formal verification toolset CADP to check the compatibility of the composition. Internally, the tool takes JSON models corresponding to the objects and the file containing the JSON bindings as input and performs a model-to-model transformation to generate the required LNT code and SVL [10] scripts for automating the analysis steps. The result of the compatibility check is returned to the user. In case of an incompatible composition, the user may want to restart the composition process. S/he can update the bindings or choose a new set of devices and check if they are compatible. Once a compatible solution is found, s/he can proceed to the deployment step.

Deployment: Finally, in the deployment step, the tool connects to Majord'Home [13], a modular IoT platform to configure the network layer. Majord'Home is a Software Defined Network (SDN) [14] based platform, which offers

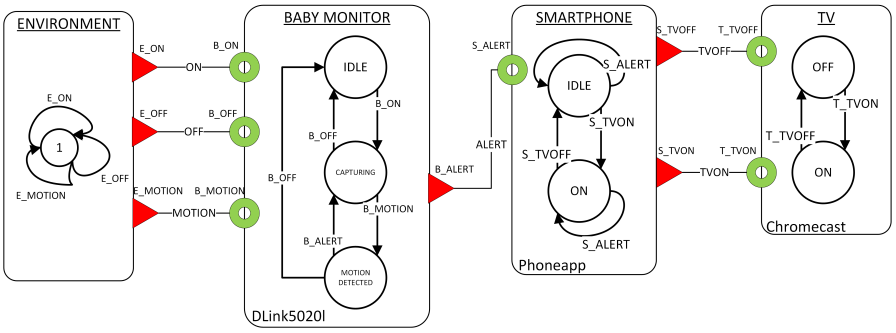

Figure 4. IoT use case showing model behaviour and bindings.

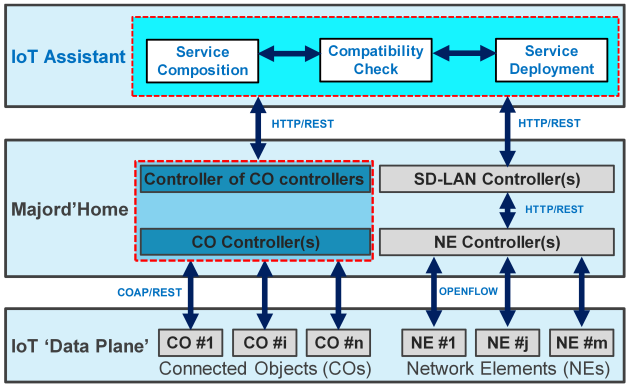

Figure 5. Tool integration with Majord'Home platform.

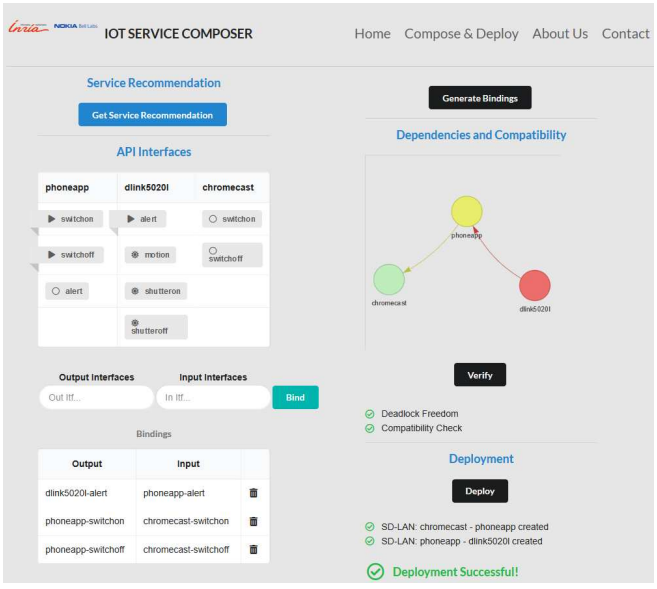

Figure 6. Screenshot of the web interface.

simplified interfaces for the network configuration of IoT services across multiple smart environments. Majord'Home is used as a deployment platform as it offers a finer degree of control to dictate the deployment leveraging specific features provided by the platform in terms of service isolation and device sharing. However, any IoT platform that provides APIs for network control can be used for deployment. Deployment is based on the plan described in Section IV. The network at SDN level is configured accordingly. In a composition, there may be one or more bindings between a pair of objects. For each set of bindings between a pair of objects, a Software-Defined LAN (SD-LAN) is set up by configuring the Residential Gateways. Using this setup, COs spread across different smart environments can communicate as they were in a same smart environment [15]. Additionally, the SD-LAN solution allows a CO to interact with different IoT services yet preserves the network isolation and discovery properties. The designed IoT assistant manages and controls SD-LANs by using custom REST APIs. The overall integration of the assistant with Majord'Home is shown in Figure 5. A CO controller is required in the architecture to facilitate the control and configuration of COs. Also, the network elements are treated as COs. This abstraction is due to the fact that there is a strong inter-working between configuration of COs and the network layer in efficiently deploying IoT services.

The tool screenshot is shown in Figure 6. Initially, the service recommendation assistant suggests the following devices: A DLink camera, an Android 7.0 tablet, and a Chromecast device. These devices can be used to setup a baby monitoring service in a smart home. In Figure 6 (top left), we can see the device list on the web interface along with available application interfaces. The output interfaces are highlighted using a ribbon with a triangle in the device list. Further in Figure 6 (bottom left), we can see the user established bindings. The dependency graph of the objects' is displayed to the user (Figure 6, top right). The compatibility checks are successful for the given scenario and also we can notice the SD-LAN creation in the screenshot (Figure 6, bottom right). More details regarding the tool and its implementation are available in the companion tool paper [16].

B. Experiments

In order to validate our approach, we considered various IoT objects available in a smart home environment and composed a number of use cases involving these objects. We focus on performance and usability aspects of the tool.

Some of the experiment results for measuring the performance are shown in Table I. The host machine used was running Xubuntu 18.04 on a hardware consisting of Core i7-7600U processor, 256GB M.2 PCIe SSD, and 16GB of RAM. In Table I, Objs denotes the number of IoT objects used in the composition (environment included). Bind refers to the number of bindings. The next two columns denote the number of states and transitions in the composition LTS. The time taken to generate the LTS in seconds (rounded off to nearest integer) and the time taken to perform the compatibility check (in seconds) are shown in the last two columns. We have not shown the deployment plan generation time as it is the time required for topological sorting, which is linear with respect to the number of objects and bindings. Moreover, it is negligible considering the context of a smart home. The time taken for deployment is negligible with respect to analysis steps as it predominantly involves establishing network connections between objects.

In Table I, the time taken to generate the LTS and check the compatibility does not proportionally increase with the number of objects and bindings, as the size of the composition LTS depends not only on the number of objects, but also on the complexity of their individual behaviours. The *Multi* prefix refers to use cases where objects are replicated in the scenario (e.g., a pair of smart doors in MultiDoor use case). We can also observe that use cases IndepCase1 and IndepCase2 involve many objects and consequently, the generated LTS is quite large. In these two use-cases, we simulated multiple instances of compositions running independently to introduce a high level of concurrency. IndepCase2 takes about 12 seconds to perform the compatibility check. This time is satisfactory considering the fact that the check is done at design time where there is no tight time constraints. To sum up, the results of the experiments confirm our understanding that the modelling and analysis approach scales well and can be used for correct IoT deployments in the context of a smart home.

In addition to performance, the tool usability was measured by involving three end-users. The users were of different ages ranging from 21 to 50 years old. Users were provided a short 20 minute walkthrough of the tool and they were explained briefly the underlying techniques (behavioural models, formal verification etc.). Generally, when carrying out the experiments we observed that an average time of 4 minutes was taken to visually build a realistic composition (select interfaces and bind) using 4 objects. Interestingly, all the users had a clear goal in mind when they began their compositions. A common feedback among the users was to reduce the number of click interactions in the tool. It is true that with the current UI, building a composition involving more than 10 objects can take a bit longer. But, as we target smart homes, where a given IoT application does not involve all the objects in the home (often less than 10 objects), we expect the composition to be built in a reasonable amount of time.

Finally, if we are to scale this approach for large IoT systems like smart cities, it would require a redesign of the UI and binding techniques. Nevertheless, the verification and deployment steps are likely to scale as the compositional verification capabilities of CADP can handle systems with hundreds of parallel components [17].

VI. Related Work

In this section, we will review the state of the art on three aspects: compatibility of behavioural models, composition-deployment of IoT applications and deployment of distributed applications.

Compatibility: Compatibility has been studied in different areas such as software architectures, software components, or Web services, but has not been the source of research yet in the IoT community. [18], [19] use an automata-based formalism as model and rely on a notion called *unspecified receptions* stating that every send message must have a corresponding receive message. [20] proposes the π-calculus as modelling language and defines a compatibility relation taking inspiration into Milner's bisimulation notion. In [21], the authors address the composability of components. They assume that two software components are composable if their respective services are pairwise compatible, where service compatibility is understood as deadlock-freeness. [22] proposes a generic framework for verifying several notions of compatibility between behavioural models. The analysis is achieved using the rewriting logic based Maude's toolset. [23] proposes an approach based on Symbolic Observation Graphs (SOG) allowing one to decide whether two web services can cooperate safely. The compatibility between two services is defined by the soundness property on open workflow nets. Compared to these works, our approach proposes a new compatibility notion in the context of IoT objects composition. The verification of this compatibility notion is fully automated using an encoding into process algebra and LTS traversal techniques.

Composition (Tools): We review recent results and tools for the composition and configuration of IoT applications. From an industrial perspective, Node-RED [24] and IFTTT [25] are two tools that provide graphical support for visually building applications consisting of IoT objects. SmartThings [26] platform provides ability to write compound rules for home automation using SmartRules app [27]. Sharp Tools [28] has a visual builder and dashboard to automate and monitor smart homes. Stringify [29] uses Flows similar to IFTTT to build automation. webCoRE [30] is a community rule engine that allows users to create scripts that are interpreted and executed by SmartThings. openHAB [31] is an open source home automation software that allows to define rules based on events, time and other triggers. These rules are executed by the openHAB engine. The home automation products in the industry help users build automation scenarios but they do not

Use case	Objs	Bind	LTS		Gen (secs)	Compat (secs)
			States	Trans.		
SmartDoor1	3	4	8	13	2	5
BabyMonitor	4	6	13	17	3	6
SmartAccess	5	8	8	9	2	5
MultiDoor	6	6	6	7	2	4
MultiCase	10	12	36	154	2	5
MultiCase2	13	15	176	892	4	9
IndepCase1	16	18	876	5134	5	11
IndepCase2	20	24	10501	79886	5	12

Table I
EXPERIMENTS SHOWING DIFFERENT IOT USE CASES.

fully take into account the dependencies among devices at the behavioural level.

Composition (Techniques): The work presented in [32] shows how to use Answer Set Programming (ASP) techniques to represent configuration scenarios for basic applications in the IoT. The authors in [33] present a formal approach for the decomposition of process-aware applications to be deployed in IoT environments. These applications are modelled using Petri nets and correctness of the decomposition is proved with respect to language preservation. In [34], the authors present a solution to the dynamic composition of services. To do so, they rely on stateful models of services, contextual information, a goal description and planning techniques in order to generate automatically a resulting composition of services. Similarly to [33], [34], we rely on behavioural models of objects. In our approach, we preferred semi-automated composition techniques (bindings required as input) to keep the user in the loop and we provide automated techniques for verifying correctness of the composition. We also propose full automation of the deployment of the final application, which makes our approach supporting the development of IoT applications from selection to final deployment.

Deployment: Finally, deployment of distributed applications has mainly been investigated in the cloud computing area. There are several configuration management tools, such as Puppet or Chef, which allow one to automatically configure new machines as described in dedicated files called manifests or recipes. Industrial technologies, such as BOSH, Cloudify, and Heat, help to create, deploy, and orchestrate applications in multiple cloud infrastructures. As far as the configuration steps are concerned, these technologies rely on the aforementioned configuration management tools (e.g., Puppet or Chef). [35] presents a formal model of components and a sound and complete algorithm for computing the sequence of actions that permits the deployment of a desired configuration even in the presence of circular dependencies among components. In [36], the authors present an extension of the TOSCA standard in order to specify the behaviour of a cloud application's management operations. They also propose several kinds of analysis for checking, e.g., the validity of a management plan or the possibility to reach a certain configuration given a plan. [37] presents a self-deployment protocol that aims at configuring a set of software components distributed over a set of virtual machines. This protocol works in a fully automated and decentralized way while supporting VM failures. Compared to these works, we propose an automated deployment solution for IoT applications, which is formally verified through abstract models and relies on SDN-based deployment.

VII. CONCLUSION

This paper has presented a solution for supporting the design and deployment of IoT applications. We first propose to support the correct construction of compositions of service-based devices by using behavioural models and automated verification techniques. Our proposed interface-based model for IoT objects, provides a generic view of the device and its behaviour, thereby making the composition agnostic to heterogeneity found across different device manufacturers. The compatibility analysis ensures that all the bindings defined in the composition can effectively be executed when the application is deployed. Once the composition is validated, a deployment plan is generated and can be executed in order to effectively configure and run the application. All the steps of our approach are supported by a prototype tool we implemented and that allows users to graphically compose, validate, and deploy IoT services. This tool has been interfaced with the Majord'Home platform and several case studies were used for validating our solution which turns out to be very helpful in practice.

Moving forward, the presented work offers several perspectives. First, our deployment approach can be extended to deal with object compositions containing cycles of strong bindings. A possible solution is to identify bindings that can be considered as weak (and hence enable the application of our current deployment scheme) and have a minimal impact on the service during its deployment. The second one relates to the extension of the presented composition framework to support the reconfiguration process (removing or adding

new devices to running applications) preserving the level of correctness ensured during the deployment phase. The final perspective relates to extending the formal models of IoT objects and composition with value-passing communication and quantitative information (e.g., probabilities, costs, etc.) to carry out QoS analysis and performance evaluation of the IoT applications.

REFERENCES

[1] L. Sanchez, L. Muñoz, J. A. Galache *et al.*, "SmartSantander: IoT experimentation over a smart city testbed," *Computer Networks*, vol. 61, pp. 217–238, 2014.

[2] A. Beugnard, J. Jézéquel, and N. Plouzeau, "Making components contract aware," *IEEE Computer*, vol. 32, no. 7, pp. 38–45, 1999.

[3] D. Champelovier, X. Clerc, H. Garavel *et al.*, "Reference Manual of the LNT to LOTOS Translator (Version 6.7)," 2018, 153 pages.

[4] H. Garavel, F. Lang, R. Mateescu *et al.*, "CADP 2011: A Toolbox for the Construction and Analysis of Distributed Processes," *STTT*, vol. 15, no. 2, pp. 89–107, 2013.

[5] M. L. Pallec, M. O. Mazouz, and L. Noirie, "Physical-interface-based IoT service characterization," in *Proc. of IOT'16*, 2016, pp. 63–71.

[6] H. Raffelt, B. Steffen, T. Berg, and T. Margaria, "Learnlib: A framework for extrapolating behavioral models," *International journal on software tools for technology transfer*, vol. 11, no. 5, p. 393, 2009.

[7] R. J. van Glabbeek and W. P. Weijland, "Branching Time and Abstraction in Bisimulation Semantics," *J. ACM*, vol. 43, no. 3, pp. 555–600, 1996.

[8] M. Ouederni, G. Salaün, J. Cámara, and E. Pimentel, "Comparator: A tool for quantifying behavioural compatibility," in *Proc. of FASE'14*, 2014, pp. 306–309.

[9] H. Garavel, F. Lang, and W. Serwe, "From LOTOS to LNT," in *ModelEd, TestEd, TrustEd - Essays Dedicated to Ed Brinksma on the Occasion of His 60th Birthday*, 2017, pp. 3–26.

[10] H. Garavel and F. Lang, "SVL: A scripting language for compositional verification," in *Proc. of FORTE'01*, 2001, pp. 377–394.

[11] R. Tarjan, "Depth-first search and linear graph algorithms," *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.

[12] M. L. Pallec, L. Noirie, P. Peloso *et al.*, "Digital assistance for the automated discovery and deployment of IoT services," in *Proc. of ICIN'18*, 2018, pp. 103–106.

[13] M. Boussard, D. T. Bui, R. Douville *et al.*, "The Majord'Home: a SDN approach to let ISPs manage and extend their customers'home networks," in *Proc. of CNSM'14*, 2014, pp. 430–433.

[14] Open Networking Foundation, "Software-defined networking: The new norm for networks," *ONF White Paper*, vol. 2, pp. 2–6, 2012.

[15] M. Boussard, D. T. Bui, L. Ciavaglia *et al.*, "Software-defined lans for interconnected smart environment," in *Proc. ITC'15*, Sept 2015, pp. 219–227.

[16] A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün, "IoT Composer: Composition and deployment of IoT applications," in *41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, Canada*, 2019.

[17] A. Bouzafour, M. Renaudin, H. Garavel, R. Mateescu, and W. Serwe, "Model-checking synthesizable systemverilog descriptions of asynchronous circuits," in *24th IEEE International Symposium on Asynchronous Circuits and Systems, ASYNC 2018, Vienna, Austria, May 13-16, 2018*, 2018, pp. 34–42.

[18] D. Brand and P. Zafiropulo, "On communicating finite-state machines," *J. ACM*, vol. 30, no. 2, pp. 323–342, 1983.

[19] D. M. Yellin and R. E. Strom, "Protocol specifications and component adaptors," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 2, pp. 292–333, 1997.

[20] C. Canal, E. Pimentel, and J. M. Troya, "Compatibility and inheritance in software architectures," *Sci. Comput. Program.*, vol. 41, no. 2, pp. 105–138, 2001.

[21] C. Attiogbé, P. André, and G. Ardourel, "Checking component composability," in *Proc. of SC'06*, ser. LNCS, vol. 4089. Springer, 2006, pp. 18–33.

[22] F. Durán, M. Ouederni, and G. Salaün, "A Generic Framework for N-protocol Compatibility Checking," *Sci. Comput. Program.*, vol. 77, no. 7-8, pp. 870–886, 2012.

[23] K. Klai and H. Ochi, "Checking Compatibility of Web Services Behaviorally," in *Proc. of FSEN'13*, ser. LNCS, vol. 8161. Springer, 2013, pp. 267–282.

[24] JS Foundation. (2018) Node-red: Flow-based programming for the IoT. [Online]. Available: https://nodered.org/

[25] S. Ovadia, "Automate the internet with If This Then That (IFTTT)," *Behavioral & social sciences librarian*, vol. 33, no. 4, pp. 208–211, 2014.

[26] SmartThings. (2018) Smartthings: Add a little smartness to your things. [Online]. Available: https://www.smartthings.com/

[27] SmartRules. (2018) Smartrules: Rule your smart home. [Online]. Available: http://smartrulesapp.com/

[28] Sharp Tools. (2018) Sharp tools: Visualize and automate your smart home. [Online]. Available: https://sharptools.io/

[29] Stringify. (2018) Stringify: Change your life by connecting every thing. [Online]. Available: https://www.stringify.com/

[30] webCoRE. (2018) webcore: The web community's own rule engine. [Online]. Available: https://wiki.webcore.co/webCoRE

[31] openHAB. (2018) openhab: Empowering the smart home. [Online]. Available: https://www.openhab.org/

[32] A. Felfernig, A. Falkner, A. Müslüm, S. P. Erdeniz, C. Uran, and P. Azzoni, "Asp-based knowledge representations for iot configuration scenarios," in *19th International Configuration Workshop*, 2017, p. 62.

[33] S. Tata, K. Klai, and R. Jain, "Formal model and method to decompose process-aware IoT applications," in *Proc. of OTM'17*. Springer, 2017, pp. 663–680.

[34] A. Bucchiarone, A. Marconi, M. Pistore *et al.*, "A context-aware framework for dynamic composition of process fragments in the internet of services," *J. Internet Services and Applications*, vol. 8, no. 1, pp. 6:1–6:23, 2017.

[35] T. A. Lascu, J. Mauro, and G. Zavattaro, "Automatic deployment of component-based applications," *Sci. Comput. Program.*, vol. 113, pp. 261–284, 2015.

[36] A. Brogi, A. Canciani, and J. Soldani, "Modelling and analysing cloud application management," in *Proc. of ESOCC'15*, ser. LNCS, vol. 9306. Springer, 2015, pp. 19–33.

[37] X. Etchevers, G. Salaün, F. Boyer *et al.*, "Reliable self-deployment of distributed cloud applications," *Softw., Pract. Exper.*, vol. 47, no. 1, pp. 3–20, 2017.