Automated Composition, Analysis and Deployment of IoT Applications

Francisco Durán[1], Gwen Salaün[2], and Ajay Krishna[2]

[1] University of Málaga, Málaga, Spain
[2] Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, 38000 Grenoble, France

Abstract. Building IoT applications of added-value from a set of available devices with minimal human intervention is one of the main challenges facing the IoT. This is a difficult task that requires models for specifying objects, in addition to user-friendly and reliable composition techniques which in turn prevent the design of erroneous applications. In this work, we tackle this problem by first describing IoT applications using abstract models obtained from existing models of concrete devices. Then, we propose automated techniques for building compositions of devices using a repository of available devices, and an abstract goal of what the user expects from such compositions. Since the number of possible solutions can be quite high, we use both filtering and ranking techniques to provide the most pertinent solutions to users. The provided solutions satisfy the given goal and may be analysed with respect to properties such as deadlock-freeness or unmatched send messages. Finally, the application can be deployed using existing execution engines.

1 Introduction

The Internet of Things (IoT) is a network of physical devices and software entities that interact together for fulfilling an overall objective. Although the devices are already available and omnipresent in our daily lives, the software allowing us to easily connect and manipulate those objects is still under development. Composition of devices and objects is a difficult and error-prone task for several reasons. First, there is a need for languages and models for describing (heterogeneous) objects or object interfaces. Several levels of expressiveness can be considered depending on the characteristics of the object (signature, behaviour, semantics, quality of service). Once a model of objects is properly defined, one can design a composition by specifying how these objects interact. This composition process should be as automated as possible to make it usable in practice by any end-user. Moreover, when building such a composition, several kinds of mismatch can arise resulting in an erroneous application. Finally, the goal is to deploy and run IoT applications with minimal human intervention.

In this paper, we propose techniques for supporting end-users during the composition and deployment tasks. We have a specific focus in this work on behavioural models for objects, that is, each object must exhibit the actions or messages it can execute as well as the order in which these actions must be triggered. Given such models, our techniques aim at automatically building satisfactory compositions given a repository of available objects and a description of the result that we call *goal*. The goal is an

abstract specification of what the user expects from the resulting composition. A composition is satisfactory if it conforms to the goal requirements. Moreover, a composition can be analysed to check whether some additional correctness properties are verified. For example, such a property can state that each reachable send message has a matching receive message in another object. Finally, when a satisfactory composition is obtained, it is deployed by relying on existing execution engines.

Our solution consists of several consecutive steps for computing satisfactory compositions. First, we extract from the repository relevant objects wrt. the goal of the composition. For those objects, we compute all combinations and apply filtering techniques for keeping only objects exhibiting expected interactions according to the goal. We then check whether the remaining candidate compositions respect a defined compatibility notion. In this paper, we use a notion based on the equivalence of the intended goal with a candidate composition of objects. Additional properties can be verified on the resulting compositions, such as, e.g., the absence of deadlocks. If there are several compositions that satisfy the goal, we use ranking techniques for presenting the results in a specific order according to several possible relevance criteria.

The whole composition process is automated by an implementation in Maude [11]. We also propose full automation of the deployment of the designed application, which allows our approach to support the development of IoT applications from the selection of a subset of satisfactory objects to their final deployment. In this work, we rely on Mozilla Project Things as execution and deployment platform, although other IoT platforms could have been used (Home Assistant, IFTTT, OpenHAB, etc.). We assume that objects are described using the Web Thing Description format by Mozilla. We also assume that the objects available in a given context (room, house, building, etc.) can be discovered using a *search* service, resulting in what we call a repository of objects. The goal provided by the end-user is described using a set of rules *IF event THEN action* representing what is expected from the generated composition.

Our prototype tool was applied to several examples for validation purposes. Since we target in this work applications at the level of a building (private house, office, nurse home, etc.), we made experiments with repositories consisting of about one hundred objects, for which the approach was able to compute compositions satisfying a given goal within a reasonable amount of time.

The rest of this paper is organized as follows. Section 2 introduces the model we use for objects and other notions (repository, goal, environment). Section 3 first presents the different steps that constitute our approach for automated composition of objects, and then describes our implementation and some experiments we carried out on several examples. Section 4 presents how IoT applications can be deployed in practice. Section 5 compares our approach to related work and Section 6 concludes the paper.

2 Models

In this section, we first introduce the model we use for describing devices and objects. Note that in the paper, for the sake of simplicity, we mainly use *object* as a common term for both devices and software elements. A repository is a set of objects, each object belonging to a family (TV, camera, light bulb, software app, window, etc.). We

```
{ "name": "Hue temp sensor",
  "type": "",
  "@context": "https://iot.mozilla.org/schemas",
  "@type": [ "TemperatureSensor" ],
  "href": "/things/hue-2",
  "properties": { "temperature": {
        "title": "Temperature",
        "type": "number",
        "@type": "TemperatureProperty",
        "unit": "degree celsius",
        "readOnly": true,
        "links": [ { "rel": "property",
        "href": "/things/hue-2/properties/temperature" } ] } } }
```

Listing 1. An excerpt from the JSON Thing Description of Hue temperature sensor

then present our notion of composition, which relies on implicit bindings. Finally, we define the notion of goal that is used in this work for guiding the composition process and the notion of environment for modelling open systems.

In this work, we assume that objects are described using Mozilla's Web Thing Description format,[3] which provides a vocabulary for describing objects in a machine readable format with a JSON encoding. The description is complementary to the current W3C Web of Things (WoT) Working Group's abstract data model. WoT architectural style uses foundations of Web technology to build IoT in a decentralised, inter-operable, and scalable fashion [15]. Our choice of WoT-based description is guided by the fact that *things* in WoT are backed by a standard data model and APIs which help in real-world deployment of objects. However, for designing the composition, we prefer to rely on an abstract model for objects, where we just keep the two most important attributes from a composition perspective, namely Event and Action. An *event* is emitted by a device (e.g., a room becomes too dark) whereas an *action* can be carried out on a device (e.g., turn on a light). The *properties* attribute in a Thing Description provides information on the type of event or action the device supports. Listing 1 shows an excerpt of the Hue temperature sensor description which describes a temperature sensor property. Since an order is possible between several events/actions (e.g., turn on a light and then turn off the light), we also keep this order in the model. To sum up, we describe an object using a Labelled Transition System (LTS) where labels either correspond to events or actions.

Definition 1. *An object is an LTS* (S, s^0, Σ, T) *where* S *is a finite set of states,* $s^0 \in S$ *is the initial state,* $\Sigma = \Sigma^! \cup \Sigma^? \cup \{\tau\}$ *is a finite alphabet partitioned into a set of events, a set of actions, and the internal action* τ*, and* $T \subseteq S \times \Sigma \times S$ *is a transition relation.*

We write $m!$ for an action $m \in \Sigma^!$ and $m?$ for an event $m \in \Sigma^?$. We also call them as send and receive messages, respectively, for homogeneity reasons. We use the symbol τ for representing internal activities (variable assignment, internal computation, local decision, etc.). A transition is represented as $s \xrightarrow{l} s' \in T$ where $l \in \Sigma$.

[3] Project Things by Mozilla: `https://iot.mozilla.org/wot/`.

This abstract model presents two advantages. First, it allows us to support any concrete model for a Thing Description in a uniform way. Second, it is simple and expressive enough for designing a composition consisting of several objects.

We assume that each object comes with a defined model that can be automatically obtained from its JSON description. As far as the behavioural part is concerned (LTS), this 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 [23].

We call *repository* a set of objects available, for example, in a room, a house or a building, depending on the context of the application. In practice, these objects are discovered using a *search* functionality that allows to identify all objects available on a given network. Search can be implemented using the mDNS protocol on the network or via Bluetooth Eddystone beacon for devices in close physical proximity. Each object is defined by its concrete model in JSON format. As far as composition is concerned, abstract models introduced before are enough, and therefore we assume that each object is defined by its abstract model (LTS) and is associated to a family.

Definition 2. *A repository is a set of couples* (F, O) *where* F *is a family name and* O *is a set of objects defined by their abstract model.*

A *composition* is defined as a set of objects described by their abstract models. We assume the objects involved in a composition interact using binary communication. This means that one interaction occurs between one send message (action) and one receive message (event) of two different objects on the same message name. Additionally, we consider a synchronous communication model, that is, two objects involved in an interaction evolve at the same time when communicating (a.k.a. handshake communication). Considering asynchronous communication (communication via message buffers or publish-subscribe) is part of the perspectives of this work.

A *goal* is an abstract description of what the end-user expects from the composition-to-be. To define this goal, we take inspiration into recent languages proposed for connecting devices and software (IFTTT) or for smart home software automation (OpenHAB). A goal is thus defined as a set of rules *IF x THEN y*, where x and y correspond to interactions between two objects. A goal is also defined by a set of family names, which gives an information about the objects to participate in the composition. E.g., the user can write that she wants three objects in the composition: a TV, a camera and a motion sensor. Family names can be derived from the *@type* member annotation of the Web Thing Description model. This annotation describes the device capabilities and required properties which is the basis for defining families. E.g., the Hue temperature sensor description in Listing 1 belongs to the TemperatureSensor family of devices.

Definition 3. *A goal is a couple* (R, FS) *where* R *is a set of rules and* FS *is a set of family names.*

Since a set of rules directly translates to an LTS (each rule transforms to a sequence of two transitions outgoing from the initial state and coming back to it), goals can also be designed as LTSs (as shown in the running example at the end of this section).

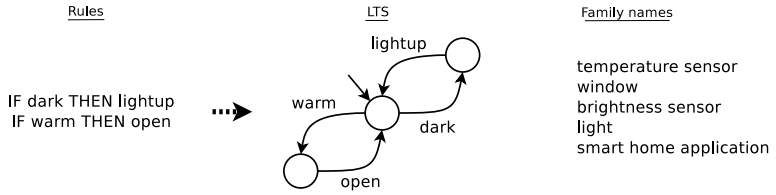

Fig. 1. Running example: goal

Last, it is worth observing that, given the dynamicity of IoT applications, we cannot assume that compositions of objects are built once and for all. They can evolve over time and objects can be added/removed for several reasons (replacement, loss of connectivity, upgrade, failure, etc.). Therefore, an IoT application can be seen like an open system where all messages are not necessarily bound and can be kept open for further addition of objects. Unbound messages can also correspond to external behaviours (a motion or an action of a human being). In our approach, we consider these actions to be part of the *environment*. The environment can be initially empty and enriched throughout the composition process, if necessary.

Definition 4. *An environment is a set of send and receive messages.*

Example. As a running example, we use a smart home application, which aims at automatically regulating the temperature and brightness in a house. To do so, we require five objects, namely a temperature sensor, a connected window, a brightness sensor, a light, as well as a piece of software, namely a smart home application running on a smartphone and acting like an orchestrator. We now define the goal of the composition using the rules (and corresponding LTS) given in Fig. 1. As described in this figure, the end-user expects two behaviours from the composition-to-be: (i) when a temperature sensor detects a too-high temperature with respect to human standards (say 20 degrees Celsius), a window should be opened; (ii) when a brightness sensor detects a too low level of luminosity, a light should be turned on. We will present in the next section the techniques we propose for automatically computing compositions satisfying the goal from a set of objects available in the repository.

3 Composition and Analysis

In this section, we present our techniques for automatically computing IoT applications by composition of available objects. Fig. 2 presents an overview of the different steps of our approach, which takes as input a repository of objects, a goal and an environment. As output, we generate a list of resulting compositions satisfying the goal (also called compatible compositions) and possibly satisfying additional properties of interest such as deadlock freeness. This list can be empty if there is no solution. If there is more than a solution, the solutions are ranked with respect to some quality criterion. Note that human intervention is required only at the beginning of the composition process to define the goal and if necessary, the environment. Each step of our approach refines the

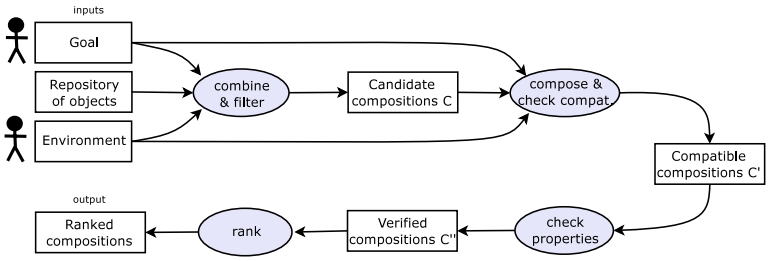

Fig. 2. Overview of the approach

number of candidate compositions (C' ⊆ C and C'' ⊆ C'), but for the last step (ranking) that only orders the compositions taken as input without discarding any of them.

In the rest of this section, we first present with more details each step of the approach. Then, we introduce the implementation of these techniques using Maude. Finally, we describe some experiments we carried out for validating our approach.

3.1 Steps of our Approach

Combine & filter. The first step of our approach takes as input the part of the goal corresponding to the set of family names and the repository, and generates all possible combinations. For example, if the composition needs a temperature sensor and a connected window, and there are two different temperature sensors and three windows in the repository, we generate all possible combinations (six possible couples in that specific case). For each combination, we apply some filtering techniques to discard it if we know beforehand that this composition will not be able to satisfy the given goal. To do so, we rely on static analysis of the alphabets of the objects involved in a candidate composition. We do not want to build the result of the composition (the LTS corresponding to all possible executions of a set of interacting objects) because this would be too costly computationally speaking. This composition LTS will be built in the next step only for candidate compositions that are not discarded by the filtering process.

The filtering process aims at traversing the alphabet of the goal and at checking whether, for each element of the goal alphabet, there are two objects in the composition with matching messages on that message, that is, one send message and one receive message with that label in two different objects. This is mandatory, otherwise no interaction would be possible on that message, resulting in a deadlock. This approach is purely syntactic, so it is very efficient, but we may still have unsatisfactory candidate compositions. The next step builds the behavioural composition for this set of candidate objects and explores all possible executions to verify that the aforementioned interactions can effectively occur according to the behavioural models of the involved objects.

Definition 5. *Given a goal* (G, FS) *with* $G = (S_G, s^0_G, \Sigma_G, T_G)$ *and a set of n object LTSs* $(S_i, s^0_i, \Sigma_i, T_i)$ *corresponding to a candidate composition according to the family names given in* FS*, this composition is not filtered out iff for each* $m \in \Sigma_G$*,* $\exists i, j \in \{1, \ldots, n\}$*, such that* $m \in \Sigma^!_i$ *and* $m \in \Sigma^?_j$*.*

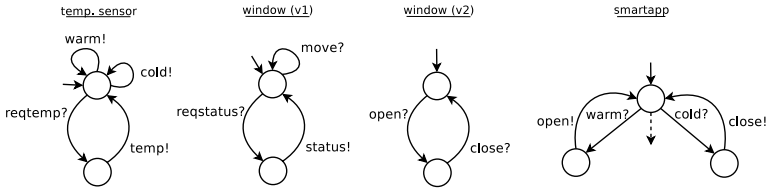

Fig. 3. Running example: filtering techniques

Example. Let us illustrate filtering techniques on the running example, and more particularly on the bottom half of the goal given in Fig. 1, which focuses on temperature and windows. In this part of the goal, we need one temperature sensor, one window and one smart app. The smart app acts like an orchestrator and is usually defined by end-users or reused from existing applications proposing common scenarios for them. The smart app may provide many functionalities, but we focus on those of interest with respect to the current goal. Fig. 3 gives four possible objects that match the required family names. Consider successively the two following combinations (temp. sensor, window (v1), smartapp) and (temp. sensor, window (v2), smartapp). The first composition is discarded by our filtering techniques, because there is one interaction appearing in the goal ('open') that is not possible in the composition (missing 'open?' message). The second composition is preserved by our filtering techniques because both 'warm' and 'open' interactions are possible.

Compose & check compatibility. This step takes as input all combinations of objects obtained using the family names defined in the goal and the repository, which have been kept after application of the filtering techniques. For each combination, we first build the resulting LTS corresponding to the composition a.k.a. synchronous product [2] in the automata-based terminology. This LTS is built independently of the goal. It considers the objects involved in the combination and the environment. All synchronize on the intersection of their alphabets (no independent evolution of observable messages). We recall that the communication model is synchronous, binary, and matches two transitions with the same label and opposite directions (sender and receiver).

Definition 6. *Given a set of n object LTSs* $(S_i, s_i^0, \Sigma_i, T_i)$*, the synchronous composition is the labelled transition system* $CLTS = (S_c, s_c^0, \Sigma_c, T_c)$ *where:*

- $S_c = S_1 \times \ldots \times S_n$
- $s_c^0 \in S_c$ *such that* $s_c^0 = (s_1^0, \ldots, s_n^0)$
- $\Sigma_c = \cup_i \Sigma_i$
- $T_c \subseteq S_c \times \Sigma_c \times S_c$*, and for* $s = (s_1, \ldots, s_n) \in S_c$ *and* $s' = (s'_1, \ldots, s'_n) \in S_c$*:*

(interact) $s \xrightarrow{m} s' \in T_c$ *if* $\exists i, j \in \{1, \ldots, n\}$ *where* $i \neq j : m \in \Sigma_i^! \cap \Sigma_j^?$ *where* $\exists\ s_i \xrightarrow{m!} s'_i \in T_i$*, and* $s_j \xrightarrow{m?} s'_j \in T_j$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \wedge k \neq j \Rightarrow s'_k = s_k$

(internal) $s \xrightarrow{\tau} s' \in T_c$ *if* $\exists i \in \{1, \ldots, n\}, \exists\ s_i \xrightarrow{\tau} s'_i \in T_i$ *such that* $\forall k \in \{1, \ldots, n\}, k \neq i \Rightarrow s'_k = s_k$

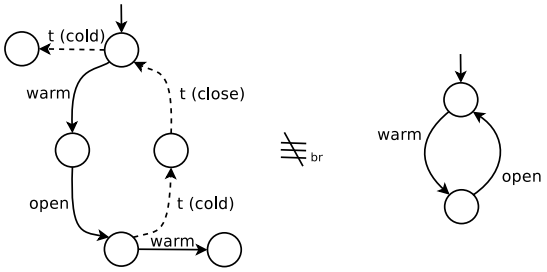

Fig. 4. Running example: composition and compatibility

Once the composition LTS is built, we first hide all interactions in the composition LTS that do not belong to the alphabet of the goal. Then, we need to compare both LTSs (goal LTS and composition LTS) to check if they produce the same observational behaviours. Since there are possibly hidden (or τ) transitions in the composition LTS, we need to use a comparison notion that takes these specific transitions into account. This is the case of the branching bisimulation [25] ($\equiv_{br}$), which is one of the finest bisimulation notions to compare LTSs in presence of hidden actions.

Definition 7. *Given a goal LTS G and a composition LTS CLTS build from a set of object LTSs, these objects satisfy the goal iff:* $hide_{\Sigma_G}(CLTS) \equiv_{br} G$.

These two steps are applied to all candidate compositions issued from the combine and filter step. Each composition respecting the above criterion is part of the resulting set of compatible compositions.

Example. Going back to the running example and particularly to the result obtained in the previous step to compose objects temp. sensor, window (v2), and smartapp (Fig. 3), we first build the composition of those three objects. We assume the environment is empty in that case. Fig. 4 shows the resulting composition LTS where we can see that there are four possible interactions in a loop. Note that this loop exists because the window can alternatively be opened and closed in sequence. Moreover, there are two other interactions ('cold' and 'warm') corresponding to messages exchanged between the temperature sensor and the smart application. Therefore, as far as compatibility is concerned, when we compare the goal (half of it to be precise) with the composition LTS where we hide interactions 'cold' and 'close' (dashed transitions) that do not belong to the alphabet of the goal, these two LTSs are not bisimilar and the candidate composition is not a compatible composition.

Assume now a slightly different version of the temperature sensor where the model actually alternates between warm and cold messages as shown in Fig. 5 (left). When composing this temperature sensor with the connected window (v2) and the smart app given in Fig. 3, we obtain the composition LTS given in Fig. 5 (middle) where only four possible interactions in a loop are maintained. This LTS turns out to be bisimilar then compatible when focusing on the goal alphabet only.

Check properties. After compatibility checking, we keep only compositions that satisfy the goal as explained beforehand. However, these compositions may not satisfy

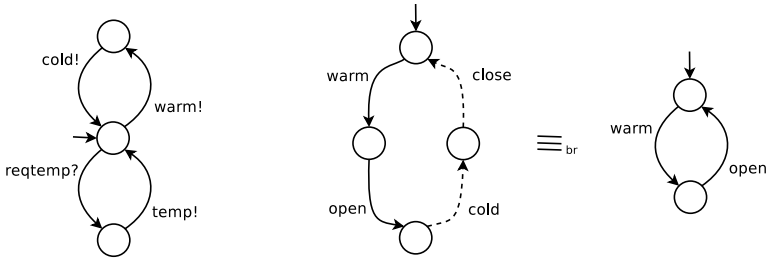

Fig. 5. Running example: (left) temperature sensor (v2) and (right) composition and compatibility

additional properties. In this paper, we focus on properties that are independent of the application. Properties that depend on the application (e.g., a specific message never occurs after another one) can be specified using temporal logic and verified using model checking techniques [4] for instance. As far as independent properties are concerned, we present two examples of such properties in this paper, namely deadlock freeness and unmatched send messages.

In our context, a deadlock occurs when there is a (global) state in the composition LTS without outgoing transition and there is one object that could evolve independently from its (local) state because in its own model there is an outgoing transition. If we put it in another way, this object has a possible behaviour to move on in its model but this transition cannot be executed in the context of the composition.

Definition 8. *Given a set of object LTSs* $(S_i, s_i^0, \Sigma_i, T_i)$ *and the corresponding composition LTS* $CLTS = (S_c, s_c^0, \Sigma_c, T_c)$*, the composition is deadlocking if there is a global state* $s = (s_1, \ldots, s_n) \in S_c$ *such that* $\nexists s \xrightarrow{l} s' \in T_c$ *but* $\exists j \in \{1, \ldots, n\}, s_j \in S_j$ *and* $s_j \xrightarrow{l'} s'_j \in T_j$ *where* l' *is either a send or receive message.*

This notion of deadlock is quite strong, because we focus on global states without outgoing transitions. There is another similar case in which there is a global state with at least one outgoing transition, and there is one object with a local transition outgoing from that state labelled with a send message that does not appear in the composition LTS. This property allows one to detect unmatched send messages.

Definition 9. *Given a set of object LTSs* $(S_i, s_i^0, \Sigma_i, T_i)$ *and the corresponding composition LTS* $CLTS = (S_c, s_c^0, \Sigma_c, T_c)$*, there is unmatched send messages if* $\exists s = (s_1, \ldots, s_n) \in S_c$*, such that* $\exists s \xrightarrow{l} s' \in T_c$ *and* $\exists j \in \{1, \ldots, n\}, s_j \in S_j$ *and* $s_j \xrightarrow{m!} s'_j \in T_j$ *and* $\forall s \xrightarrow{l} s' \in T_c, l \neq m$*.*

Example. For illustration purposes, let us focus on the part of the running example that aims at lighting up/down the room depending on the level of brightness. To do so, we present three objects in Fig. 6 (left) resulting in a compatible composition: the composition LTS (middle) and the goal (right) are bisimilar. The resulting composition (middle) is free of deadlocks, but is not free of unmatched send messages because in both global states in the composition, the brightness sensor can always send a 'bright!' message, which is not caught in the composition because there is no counterpart ('bright?') in

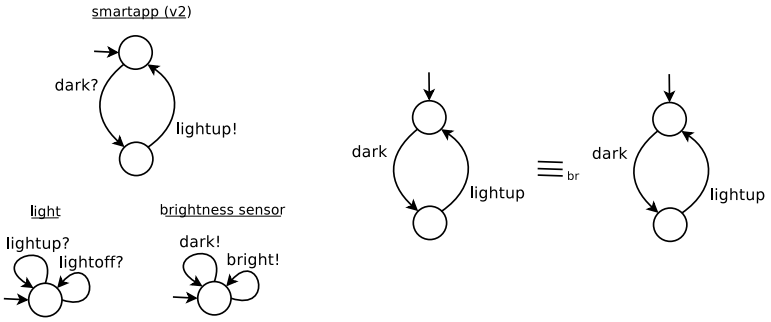

Fig. 6. Running example: (left) objects, (middle) composition and (right) goal

any other object. In such a situation, the user can either decide to move on because this is not a problem (can be amended, for example, by adding 'bright?' to the environment) or to choose another compatible composition satisfying this property.

Rank. In its current version, we rank satisfactory compositions according to their *complexity*. As complexity criterion, we consider the size of the composition LTS in terms of number of transitions. The composition ranked first corresponds to the solution that can satisfy the expected goal with the lowest number of interactions.

3.2 Implementation

We have developed the techniques presented in the former section using Maude's rewriting logic framework [11]. We chose Maude for implementing the composition techniques because its declarative style facilitates program writing, and specifically, it is quite simple to implement filtering techniques, composition or compatibility analysis. Moreover, Maude is adequate to specify concurrent systems and is equipped with a large variety of analysis tools.

Maude is a high-level language and a high-performance system that supports membership equational logic, and rewriting logic specification and programming of systems. Rewriting logic [20] is a logic of change that can naturally deal with state and with highly nondeterministic concurrent computations. Rewriting logic is parameterised by an equational logic, and therefore, Maude integrates an equational style of functional programming with rewriting logic computation. In the Maude implementation of rewriting logic, the equational logic is membership equational logic [7]. Membership equational logic is a Horn logic whose atomic sentences are equalities $t = t'$ and membership assertions of the form $t : S$, stating that a term t has sort S. Such a logic extends order-sorted equational logic, and supports sorts, subsort relations, subsort polymorphic overloading of operators, and the definition of partial functions with equationally defined domains. Further details may be found in [11].

The Maude's implementation consists of different modules. Unfortunately, for the sake of space, we cannot present in detail the contents of these modules. The interested reader should look at the Maude specification, together with a set of examples, which is available online [13].

```
1  ---- seeks a global state corresponding to a deadlock
2  op deadlock : LTS Set{Device} -> Bool .
3  ---- checks whether there is a transition outgoing of that state
4  op deadState : State Set{Transition} -> Bool .
5
6  eq deadlock(model(St, empty, empty), Devices) = true .
7  ceq deadlock(
8          model(St, (s((DId |-> St1, ISM)), States), Transitions),
9          (dev(DId, model(St2, States1, ((St1 - I -> St3), Transitions1))), Devices))
10     = true
11     if deadState(s((DId |-> St1, ISM)), Transitions) .
12 eq deadlock(Mod, Devices) = false [owise] .
13
14 eq deadState(St, (St - I -> St1, Transitions))
15   = false .
16 eq deadState(
17          s((DId |-> St, ISM)),
18          (s((DId |-> St, ISM)) - Str -> s((DId |-> St1, ISM1)), Transitions))
19   = false .
20 eq deadState(St, Transitions) = true [owise] .
```

Listing 2. Equations for deadlock checking in the Maude implementation

A first module defines all necessary data types presented in Section 2 (object model, repository, goal, environment). Then, we have one module for each step of the approach presented in Section 3.1 and summarized in Fig. 2. Finally, we use a couple of additional modules for defining concrete objects grouped in a repository and several examples of goals for making experiments that we will present in Section 3.3.

Let us illustrate our Maude implementation with the verification of compositions. Listing 2 gives the equations used for identifying that a (global) state corresponds to a deadlock state. The deadlock operation takes as input the composition LTS and the set of device models. The first equation (line 6) applies when the composition LTS does not contain any state. The second equation is the most interesting. It says that if there is one global state (line 8) in the composition LTS corresponding to a deadlock state (line 11) that is a state without outgoing transition, and if there is one device with one possible transition from that local state (line 9), then this means that this behaviour cannot be executed in the context of this composition, and the equation returns true (line 10). The deadState operation (lines 14–20) takes a state and an LTS as input and checks whether from that state there is an outgoing transition. The first equation (lines 14–15) applies to simple devices. The second equation (line 16–19) applies to composition LTSs. Both equations correspond to the case in which there is such a transition and return false. The final equation (line 20) applies otherwise and returns true.

3.3 Experiments

The final part of this section presents some experiments we carried out to see how our approach scales with respect to the number of objects available in the repository and with respect to the size of the goal. The experiments were run on a macOS Mojave machine with a 2.8 GHz Intel Core i7 processor, 16GB of DDR3 RAM and 256GB PCIe-based flash storage. We recall that our approach targets small to medium-size applications, corresponding to a number of objects available in a smart home or building.

Goal			|Repo.|	|Combinations|		Time (sec.)			
Ident.	|O|	|T|		combine	filter	compo/check	combine	filter	compo	check
G1	3	6	30	160	1	1	~ 0	~ 0	~ 0	~ 0
G1	3	6	80	220	1	1	~ 0	~ 0	~ 0	~ 0
G1	3	6	150	5,544	13	12	~ 0	~ 0	~ 0	~ 0
G2	5	4	30	288	18	1	~ 0	~ 0	~ 0	~ 0
G2	5	4	80	217,800	5,760	2	1.7	3.8	3.1	5.9
G2	5	4	150	401,544	11,000	2	3.3	8.3	14.8	10.9
G3	5	8	30	7	0	0	~ 0	~ 0	~ 0	~ 0
G3	5	8	80	119	0	0	~ 0	~ 0	~ 0	~ 0
G3	5	8	150	218	1	1	~ 0	~ 0	~ 0	~ 0
G4	8	10	30	336	0	0	~ 0	~ 0	~ 0	~ 0
G4	8	10	80	143,990	220	0	19.9	22.1	~ 0	~ 0
G4	8	10	150	374,088	220	0	54.5	49.1	~ 0	~ 0
G5	10	10	150	1,800	1	1	~ 0	0.1	~ 0	~ 0
G6	15	10	150	57,600	2	2	1.1	4.7	0.3	1.6
G7	20	20	150	225,792	1	1	5.9	28.5	0.1	12.6

Table 1. Experimental results (~ 0 for values smaller than 0.1 sec.)

Table 1 presents the results obtained for seven different goals. G2 corresponds to the goal of the running example given in Fig. 1. For the first four goals, we vary the size of the repository, that is, the number of objects available in the repository. More specifically, we use three repositories of different sizes (30 objects, 80 objects, and 150 objects). As for the three last goals (G5 to G7), we only use the largest one (150 objects). Table 1 then gives the number of possible combinations according to the list of family names given in the goal (combine), the number of compositions selected after application of the filtering techniques (filter), the number of compatible compositions satisfying the goal (compose and check compatibility), and the time it takes to compute all these steps. When a time is smaller than 0.1 sec., we use ~ 0.

First of all, it is worth noting that for all these small and medium size (yet realistic) compositions, the results are computed in a reasonable time (about a total of four minutes for all the examples given in Table 1).

The increase in terms of computational time mainly comes from the number of possible combinations. It takes more time to compute and explore all possible combinations to see whether they are possible candidates (kept after filtering) and finally solutions wrt. the expected goal. The number of combinations augments for three reasons. The first factor is the number of objects in the repository. When looking at G2 in the table for instance, one can see that the number of combinations and the computation times increase when considering a repository with 30, 80 or 150 objects. The second factor is the number of objects involved in the composition. Goals G5, G6 and G7 show compositions involving 10, 15 and 20 objects. Here again, the number of objects induces a larger number of combinations and an increase in computation time. The third reason is not obvious because it concerns the number of objects in each family. If we look at goals G2 or G4 in the table with a repository of 150 objects, we can see that even

for a limited number of objects (5 and 8), the number of combinations is rather high. This is because in both cases, the families of objects specified in the goal have many instances of objects in the repository (more than 10 concrete objects for each family), resulting in many combinations. In contrast, if the goal has many objects, but families with fewer instances, as it is the case for goal G5 for example, there are not so many combinations.

Once all combinations are computed, filtering is applied on all those combinations. Obviously, the more combinations, the costlier is filtering. This is why these two numbers (time for generating combinations and for applying filtering) are related and usually quite close. The final step computes the synchronous composition of all objects for each remaining combination and checks whether the resulting composition LTS matches the goal. This step is definitely the costliest. As an example, for G2 (150 objects), after filtering there still are 11,000 possible candidates, and it takes about 25 secs. to generate the compositions and analyse them, whereas it takes less than 10 secs. to generate all combinations (about 400,000) and apply filtering on them.

As a conclusion, it is worth noting the importance of the filtering techniques that can avoid the unnecessary and costly computation of some candidate compositions as well as their compatibility analysis. All the times given in the table are reasonable because the filtering techniques return a low number of candidate compositions.

4 Deployment

This section details the real-world deployment of IoT devices using an execution platform. We use the Mozilla Project Things as it is open-source and one of the feature-rich implementations of WoT. However, any standard IoT platform, such as OpenHAB or Home Assistant, can also be used for deployment. Specifically from our point of interest, Things Gateway from Mozilla has a unified Web interface to monitor and control devices. It also provides REST APIs to create and deploy rules.

Given a set of available objects (repository) and a set of abstract rules (goal), after the application of the process presented in Section 3, we obtain a subset of these objects satisfying the rules. This section explains how this application can be deployed on the platform. Recall that the events and actions in the abstract rules are associated with the *properties* attribute of the Thing Description. Therefore, in order to deploy each abstract rule, we first map the events and actions of the abstract rules to its corresponding *properties* in the JSON Thing Description. An action or an event relates to a change in property values. Using this idea, we generate a rule in JSON format for each abstract rule. Further, we use the gateway API provided by the platform, which takes a JSON rule as input, to deploy each newly created rule.

Example. We have deployed our running example using the following devices: Philips Hue Play Light, Philips Motion Sensor which has a built-in ambient light sensor and a temperature sensor. As our device repository did not have smart window, we created virtual Things adapters to emulate the smart window. We use the *TemperatureProperty* of the Philips Hue temperature sensor as shown in Fig. 1 to build the concrete rule “IF warm THEN open". The deployable rule is shown in Listing 3, where we notice that warm translates to temperature greater than 20 degrees Celsius (*value* attribute set to

```
{ "enabled": true,
  "trigger": { "type": "MultiTrigger",
    "op": "AND",
    "triggers": [ { "type": "LevelTrigger",
      "label": "Temperature",
      "property": { "type": "number", "thing": "hue-2", "id": "temperature", "unit
           ": "degree celsius" },
      "value": 20,
      "levelType": "GREATER"} ] },
  "effect": { "type": "MultiEffect",
    "effects": [ { "type": "SetEffect", "label": "On/Off",
      "property": { "type": "boolean", "thing": "virtual-env-1", "id": "on" },
      "value": true } ] } }
```

Listing 3. WoT rule JSON corresponding to abstract rule IF warm THEN open

20). Users can configure such constants or keywords to make the rules closer to the natural language. The action in the rule is described in the *effect* attribute, where a boolean property is set to *on* in the virtual environment thing.

5 Related Work

We discuss in this section some relate work on automated composition of Web services, compatibility of behavioural models, and (automated) composition of IoT objects.

Automated composition was mainly studied in the Web services area. Several papers have been published on that problem, see, e.g., [21,5,18,6]. Most of these techniques rely on Web service languages, namely WSDL and BPEL, whereas we preferred to rely on generic behavioural models to make our solution more easily reusable in other application areas. These papers make use of existing planning techniques and tools. We preferred a different solution since we choose rule-based programming and rewriting logic for computing the resulting compositions. Our approach also provides (automated) verification techniques to check for compatibility and other properties of interest.

As far as compatibility checking is concerned, several works have focused on this problem assuming that entities are described using behavioural models, see, e.g. [8,26,10,19,3,12,17]. [10] proposes the π-calculus as modelling language and defines a compatibility relation taking inspiration into Milner's bisimulation notion. [19] presents a framework for modelling Web service with Petri nets and for analysing several properties on these models, the most important being the usability property, which is verified using the soundness criterion for workflow modules. In [3], 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. [17] proposes an approach based on Symbolic Observation Graphs (SOG) allowing one to decide whether two services can cooperate safely. The compatibility between two services is defined by the well-known soundness property on open workflow nets. Our approach proposes a notion of goal-based compatibility for IoT applications obtained by composition of available devices and its verification using rewriting logic and Maude's framework.

We finally introduce recent results and tools for the composition and configuration of IoT applications. From an industrial perspective, Node-RED [16] and IFTTT [22]

are two tools that provide graphical support for visually and manually building applications consisting of connected objects. We chose full automation and synthesis of the object composition in our approach. [14] shows how to use Answer Set Programming (ASP) techniques to represent configuration scenarios for basic applications in the IoT. [1] proposes an approach that makes a set of things connect and cooperate temporarily to achieve a user goal. [24] presents 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 [9], 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 [24,9], we rely on behavioural models of objects with a specific focus here on the automated composition of objects. We provide additional techniques for facing the large number of solutions (filtering and ranking) as well as automated verification techniques for ensuring compatibility and additional properties. Last but not least, we also support the deployment of concrete applications.

6 Concluding Remarks

We have presented some automated techniques for generating and deploying satisfactory compositions given an abstract goal of the composition-to-be and a set of object families. Our approach works applying successively different steps. First, from a repository of available objects, we generate a set of candidate compositions statically filtering those that cannot satisfy the goal. Then, we check if these candidates satisfy the given goal and are therefore compatible compositions. The user can also decide to verify additional properties that are independent of the application such as deadlock freeness or the absence of unmatched send messages. If there is more than one solution, we rank them according to some complexity criterion. Finally, the resulting composition is deployed using an existing execution engine. The composition and analysis process are supported by an implementation in Maude. The deployment process is carried out using Mozilla's Project Things platform. We applied the whole approach on several case studies for validation purposes.

The first perspective of this work is to consider location of objects in the object model and composition goal. We believe that this information would allow us to make our approach scalable in order to target larger applications (at the level of a campus or a city for example). More precisely, this information would be used during the composition process for improving our filtering techniques and for finding the best combinations without exhaustively producing and analysing all of them. Considering asynchronous communication (communication via message buffers or publish-subscribe) is another perspective of this work.

References

1. F. Alkhabbas, M. D. Sanctis, R. Spalazzese, A. Bucchiarone, P. Davidsson, and A. Marconi. Enacting Emergent Configurations in the IoT Through Domain Objects. In *Proc. of ICSOC'18*, volume 11236 of *LNCS*, pages 279–294. Springer, 2018.

2. A. Arnold. *Finite Transition Systems - Semantics of Communicating Systems*. Prentice Hall, 1994.
3. C. Attiogbé, P. André, and G. Ardourel. Checking Component Composability. In *Proc. of SC'06*, volume 4089 of *LNCS*, pages 18–33. Springer, 2006.
4. C. Baier and J. Katoen. *Principles of Model Checking*. MIT Press, 2008.
5. D. Berardi, D. Calvanese, G. De Giacomo, M. Lenzerini, and M. Mecella. Automatic Composition of E-services That Export Their Behavior. In *Proc. of ICSOC'03*, volume 2910 of *LNCS*, pages 43–58. Springer, 2003.
6. P. Bertoli, M. Pistore, and P. Traverso. Automated Composition of Web Services via Planning in Asynchronous Domains. *Artif. Intell.*, 174(3-4):316–361, 2010.
7. A. Bouhoula, J.-P. Jouannaud, and J. Meseguer. Specification and Proof in Membership Equational Logic. *Theoretical Computer Science*, 236(1):35–132, 2000.
8. D. Brand and P. Zafiropulo. On Communicating Finite-State Machines. *J. ACM*, 30(2):323–342, 1983.
9. 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*, 8(1):6:1–6:23, 2017.
10. C. Canal, E. Pimentel, and J. M. Troya. Compatibility and Inheritance in Software Architectures. *Sci. Comput. Program.*, 41(2):105–138, 2001.
11. M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *All About Maude*, volume 4350 of *LNCS*. Springer, 2007.
12. F. Durán, M. Ouederni, and G. Salaün. A Generic Framework for N-protocol Compatibility Checking. *Sci. Comput. Program.*, 77(7-8):870–886, 2012.
13. F. Durán and G. Salaün. A Note on Automated Composition, Analysis and Deployment of IoT Applications. `http://maude.lcc.uma.es/iotcompo`, April 2019.
14. A. Felfernig, A. Falkner, A. Müslüm, et al. ASP-based Knowledge Representations for IoT Configuration Scenarios. In *Proc. of ICW'17*, page 62, 2017.
15. D. Guinard, V. Trifa, F. Mattern, and E. Wilde. From the Internet of Things to the Web of Things: Resource-oriented Architecture and Best Practices. In *Architecting the Internet of Things*, pages 97–129. Springer, 2011.
16. JS-Foundation. Node-RED: Flow-based Programming for the IoT, 2018.
17. K. Klai and H. Ochi. Checking Compatibility of Web Services Behaviorally. In *Proc. of FSEN'13*, volume 8161 of *LNCS*, pages 267–282. Springer, 2013.
18. A. Marconi, M. Pistore, and P. Traverso. Automated Composition of Web Services: the ASTRO Approach. *IEEE Data Eng. Bull.*, 31(3):23–26, 2008.
19. A. Martens. Analyzing Web Service Based Business Processes. In *Proc. of FASE*, volume 3442 of *LNCS*, pages 19–33. Springer, 2005.
20. J. Meseguer. Conditional Rewriting Logic as a Unified Model of Concurrency. *Theoretical Computer Science*, 96(1):73–155, 1992.
21. S. Narayanan and S. A. McIlraith. Simulation, Verification and Automated Composition of Web Services. In *Proc. of WWW'12*, pages 77–88. ACM, 2002.
22. S. Ovadia. Automate the Internet with "If This Then That"(IFTTT). *Behavioral & Social Sciences Librarian*, 33(4):208–211, 2014.
23. H. Raffelt, B. Steffen, T. Berg, and T. Margaria. LearnLib: A Framework for Extrapolating Behavioral Models. *STTT*, 11(5):393, 2009.
24. S. Tata, K. Klai, and R. Jain. Formal Model and Method to Decompose Process-Aware IoT Applications. In *Proc. of OTM'17*, pages 663–680. Springer, 2017.
25. R. J. van Glabbeek and W. P. Weijland. Branching Time and Abstraction in Bisimulation Semantics. *J. ACM*, 43(3):555–600, 1996.
26. D. M. Yellin and R. E. Strom. Protocol Specifications and Component Adaptors. *ACM Trans. Program. Lang. Syst.*, 19(2):292–333, 1997.