Consistent Substitution of Object in Rule-based IoT Applications

Gwen Salaün
Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG, F-38000 Grenoble, France

***Abstract*—The Internet of Things (IoT) is a network of physical devices and software entities that interact together for fulfilling an overall objective. Such applications are built by selecting and composing several objects. Recent frameworks promote the use of '*if event(s) then action(s)*' rules to make explicit the way these objects interact together, i.e., if an event is raised, then an action is triggered. IoT applications are not monolithic applications built once and for all. In this paper, we focus on the replacement of an object, operation which is often required for substituting an out-of-order or obsolete device. When substituting an object by another one, the user may want the application to provide at least the same functionalities as before. Therefore, replacement should be supported by automated techniques and tools in order to guarantee the preservation of the application behaviour. As a result, we first define several notions of object substitution. Then, we show how these notions can be automatically checked or computed. Finally, we present the tool support and its integration to the Mozilla WebThings platform for applying our approach on smart home applications.**

***Index Terms*—Internet of Things, substitution, automated techniques, tools, WebThings platform.**

I. Introduction

The Internet of Things (IoT) describes the network of physical objects that are embedded with sensors, software, and other technologies for the purpose of connecting and exchanging data with other devices and systems over the internet. The IoT ecosystem is rapidly growing and it is expected that ∼125 billion connected IoT devices are to be deployed by 2030 [1]. However, the IoT comes with its number of challenges involving issues such as heterogeneity, reliability, security, privacy or maintenance. In this work, we particularly focus on the design and evolution over time of IoT applications. Indeed, IoT applications are not monolithic applications built once and for all. In contrast, they are constantly modified due to removal, replacement or addition of new objects during the application lifetime. The evolvability of the application must be taken into account in the overall design process. Automation is also required when dynamically reconfiguring IoT applications to simplify this task for end-users.

Our goal in this paper is to propose new techniques to support the possible reconfiguration of a running application. We focus here on a specific situation in which the user wants to replace one object present in the current running application by another object. This can occur for instance to replace a failed object, to replace an obsolete device by a more recent one, or to upgrade a software component. In most cases, in this situation, the user does not want to change the observable behaviour of the whole application. Instead, (s)he would like this replacement to be as seamless as possible, preserving all the existing functionalities of the application. This is not a simple task because this requires to precisely analyse or compare both objects (the current and the new one) to be sure that the replacement will preserve the application consistency.

In this paper, we present some techniques for supporting the substitution of an object by a new one in a running IoT application. We will present three possible solutions to compare both objects and thus say whether the replacement is possible or not. It is worth noting that our approach makes sense only if the two objects share some similarity from a behavioural perspective. If the user tries to replace a motion sensor by a temperature sensor for instance, this will be discarded by our approach. However, imagine that a user wants to replace an old-fashioned motion sensor by a modern one (with more functionalities), (s)he may want to be sure that the application will still be functional with that new one and will behave at least as before replacement.

In this work, we focus on IoT applications in the context of smart homes and smart buildings. Recent frameworks such as IFTTT [2], Node-RED [3] and Mozilla WebThings [4] promote the design of IoT applications by using '*if event(s) then action(s)*' rules, i.e., if an event is raised, then an action is triggered. The composition or orchestration language we consider in this work relies on such rules, but is more expressive since it allows the writing of composition of rules using classic constructs such as the sequence of rules, the choice between several rules, the concurrent execution of several rules or the repetition of rules. Given an application described using a set of objects and a composition expression written with this language, we propose three techniques for supporting the replacement of an object present in the running application by a new one. The first one relies on the comparison of both objects using verification techniques. The second technique aims at embedding the new object within a virtual object whose role is to mimic the behaviour of the replaced object. The third technique relies on rewriting of the composition expression in order to seamlessly substitute all occurrences of the former object by the new one. All these solutions are supported by several tools, some of them already existing and reused in this work, and other ones we implemented ourselves for automating specific tasks of our approach. These tools were integrated to the Mozilla WebThings platform [4], which allowed us to apply our substitution techniques on real smart home applications for validation purposes.

The rest of this paper is organised as follows. Section II introduces the model of objects and the rule-based composition language. Section III presents our new techniques for substitution of objects in IoT applications. Section IV describes the tool support and particularly the extensions of the WebThings platform to support substitution. Section V surveys related work and Section VI concludes the paper.

II. Models

An IoT application consists of a set of IoT objects or *things* interacting all together to fulfil a certain overall goal. Given the heterogeneity of devices and platforms existing in the IoT ecosystem, there is a need for a standard description format for objects. Although several proposals have recently been made, e.g., Constrained RESTful Environments [5], OpenWeave [6], Thing Description in Web of Things (WoT) [7] framework, etc., there is no widely accepted standard yet. Therefore, in this work, we prefer to rely on an abstract model for modelling objects, with a specific focus on the behavioural part of this model.

More precisely, an object is modeled as a set of events received by the object and actions emitted by the object. Since the events/actions involved in an object are executed in a specific order, we describe them using a behavioural model (Labelled Transition System or LTS). We use a question mark (?) and an exclamation mark (!) to indicate that the object is receiving or emitting from/to its environment, respectively.

Definition 1 (Behavioural Model): An IoT object is described by its behavioural model, which is a tuple (S, A, T, s^0), where S is a set of states, A is a set of events/actions associated with transitions, $T \subseteq S \times A \times D \times S$ is the transition relation where $D = \{!, ?\}$ and $s^0 \in S$ is the initial state. A transition $(s_1, e, d, s_2) \in T$ (also noted $s_1 \xrightarrow{ed} s_2$) indicates that the system can move from state s_1 to state s_2 by performing an event/action named e in a certain direction (! for sending, ? for receiving).

An IoT application in this work is described by a set of objects and a composition expression. This expression acts as an orchestrator and describes how the involved objects interact together. We use a simple rule-based composition language to this purpose. This language assumes '*if event(s) then action(s)*' rules as basic elements. A rule is triggered when one or several events are issued by specific objects and, as a reaction, one or several actions are issued to other objects defined as target. Each event or action is accompanied with its object identifier.

Definition 2 (Rule): Given a set of objects $\{O_1, \ldots, O_n\}$, $O_i = (S_i, A_i, T_i, s_i^0)$, a rule R is defined as "**IF** EVT **THEN** ACT" where,

EVT ::= $event$ (Oid) | $\text{EVT}_1 \wedge \text{EVT}_2$ | $\text{EVT}_1 \vee \text{EVT}_2$,

ACT ::= $action$ (Oid) | $\text{ACT}_1 \wedge \text{ACT}_2$,

where $event$ and $action$ are terminal symbols with $event, action \in \bigcup_{i=1}^{n} A_i$, and Oid is an object identifier.

These rules can be composed to build more complex expressions, using basic operators such as sequence, choice, concurrent execution or repetition of rules.

Definition 3 (Composition Language): A composition C is an expression built over a set of rules R using the following operators:

$$C ::= R \mid R\,;\,C \mid R + C \mid R \,||\, C \mid C^k$$

where $R\ ;\ C$ represents a rule followed by a composition expression, $R + C$ represents a choice between a rule and a composition expression, $R \,||\, C$ represents the concurrent execution of a rule and a composition expression, and C^k represents the execution k times of a composition expression (if $k = *$, C executes infinitely).

Let us now explain how the IoT application consisting of a set of objects and a composition expression executes. Each object is equipped with an input message buffer (FIFO). The composition expression and all objects start their execution from their initial states. An application can evolve in two cases: execution of a rule or buffer consumption. In the first case, if all events appearing in the left part of the rule have been issued, the rule can be triggered and all the actions appearing in the right part of the rule are pushed to the corresponding buffers of the objects. The events can occur as a result of changes in the physical environment (e.g., change in temperature) or by interacting directly with the objects (e.g., a user pressing a switch). In the second case, one object can individually consume from its input buffer, if there is something in its buffer and the object can make this move according to its behavioural model.

Example. We illustrate with an IoT application consisting of one switch and one light (Fig. 1). We can see that labels on transitions correspond to actions executed in that case by the object once they are available in its local buffer. The composition expression consists of two rules. When the switch is pressed on, the light is turned on (rule R1). Rule R2 is used for turning off the light. The composition expression indicates that both rules apply infinitely (*) in sequence (;).

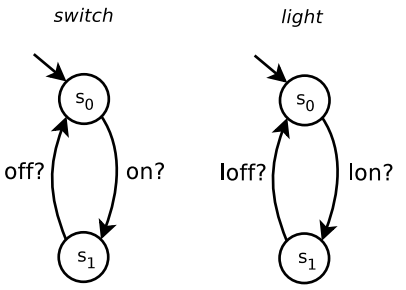


Figure 1. Example of IoT Application Model

III. Substitution

In this section, we first present several definitions of substitution. These definitions aim at preserving the original behaviour of the application. In a second step, we show how

we can compute a mapping or correspondence between the actions of the former and new object, which serves as basis to two of these substitution definitions.

A. Three Notions of Substitution

In this work, an IoT application consists of a set of objects (described using their behavioural models) and of a composition expression. We focus in this paper on the replacement of one object by another one. We will see that this replacement may impact the composition expression. Now we present three definitions of replacement.

Simulation. First of all, we propose to rely on existing notions available in concurrency theory. Trace equivalence [8] ensures that an object produces the same traces as another one, and could be considered as a first solution. This notion of equivalence is usually too weak and do not prevent for instance the addition of deadlocks. If we want to reproduce the same behaviour, a better notion is bisimulation [8], which is a strong notion ensuring that a behavioural model or LTS exactly reproduce the same behaviour without generating additional erroneous behaviours such as deadlocks. In some cases, bisimulation may appear too strong because a new object replacing an old one can reproduce the same behaviour but can also exhibit new functionalities. This is the case in IoT for instance if one wants to replace a classic light by a colored one. The former one can be turned on and off successively whereas the new one in addition can change its color. In such a case, the new one is fine because it reproduces the original behaviour but provides new features. Therefore, we think that a more adequate notion of substitution in this context is the notion of *simulation preorder* [8], which checks whether the new object simulates the original one. Intuitively, the new object can execute the same behaviour as the former object, but it can also produce additional behaviour.

Definition 4 (Strong Simulation): A relation R is a strong simulation between states in S iff for all $s_1, s_2 \in S$ such that $R(s_1, s_2)$, $(\forall l \in A, s_1' \in S)\ (s_1, l, s_1') \in T \implies (\exists s_2' \in S)\ (s_2, l, s_2') \in T \wedge R(s_1', s_2')$. Two states s_1 and s_2 are strongly similar (written $s_1 \sim_s s_2$) iff there exists a strong simulation R such that $R(s_1, s_2)$. Two LTS $M_1 = (S_1, s_1^0, A_1, T_1)$ and $M_2 = (S_2, s_2^0, A_2, T_2)$ are strongly similar (written $M_1 \sim_s M_2$) iff $s_1^0 \sim_s s_2^0$.

Example. Let us illustrate with a simple example (Fig. 2) consisting of an IoT application where a light (light1) is replaced by a new one (light2). The new object respects the simulation definition and can thus reproduce all the behaviours possible in the former object. More precisely, state s_0 is simulated by state s_0' and state s_1 is simulated by state s_1'. Note that the new light also exhibits additional behaviour, specifically the possibility to change its color.

Virtual object. The second solution aims at changing an object without changing the rest of the application (that is, the other objects or the composition expression). To do so, we propose to use a piece of software known as *virtual object*, which embeds the new object and behaves from an external point of view as the replaced object. This means that there is a mapping between the actions possible in the replaced object and the new object. More precisely, a mapping defines a correspondence between a set of actions in one LTS with another set of actions in another LTS. In its simplest occurrence, a correspondence consists of one action in the first LTS and a second one in the second LTS. All actions of each LTS alphabet must appear in the mapping. We will explain how this mapping is computed in Section III-B of this paper.

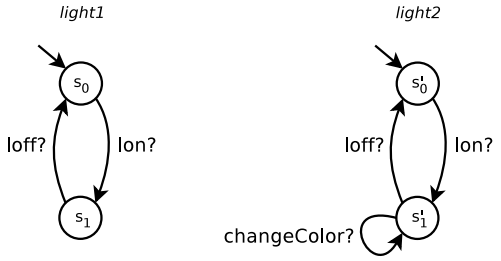


Figure 2. Example for Simulation based Substitution

Definition 5 (Mapping): A mapping between two alphabets A_1 and A_2 is defined as a set of couples $\{(set1_1, set2_1), \ldots, (set1_n, set2_n)\}$ where for $i \in 1..n$, $set1_i \subseteq A_1$, $set2_i \subseteq A_2$, $\cup set1_i = A_1$, and $\cup set2_i = A_2$.

The virtual object relies on this mapping to mimic the behaviour of the replaced object from an external point of view. To do so, when an action is issued for instance by the embedded object, and there is a match in the mapping for this action, the virtual object submits the corresponding action to the environment. If there is no match in the mapping, this action is stored. When a second action is issued by the object, there are now two actions, and the virtual object seeks for a corresponding match with these two actions or just one of them. All combinations of pending actions are possible options. Note that, when there are several possible matches, the virtual object triggers one of them in a non-deterministic way. This algorithm works similarly if the input actions are issued by the embedded object or come from the environment, but both sets of actions are handled separately.

Definition 6 (Virtual Object): A virtual object VO is a software that takes an object O and a mapping M as input, and behaves as follows:

- for each action (or set of actions) issued by O, VO submits the corresponding action(s) in M to the environment;
- for each action (or set of actions) coming from the environment, VO transmits to O the correponding action(s) in M.

Example. The example in Figure 3 shows several versions of a switch. Suppose we want to replace a classic rocker switch ($switch1$) by one with a single button ($switch2$) where it alternates on and off by simply tapping the button. To embed the second switch, still preserving from a functional point of view the behaviour of the first switch, we need the following mapping: $\{(on, tap), (off, tap)\}$. We remove the set notation

within each couple because in this case we have a one-to-one action correspondence. The virtual object keeps receiving on and off actions from the environment, and transforms them internally to tap actions. As a result, the behaviour of this object and of the whole application is exactly the same as before.

Suppose now that we want to replace the first switch ($switch1$) with the third one ($switch3$), which works like the second one by tapping a single button. However, the behaviour differs in the sense that one tap is enough for switching on, but two taps are necessary for switching off. To embed the third switch while preserving the functional behaviour of the first switch, we need the following mapping: $\{(on, on), (off, \{tap1, tap2\})\}$. In that case, once an off action comes from the environment, two actions are executed on the new device, namely $tap1$ and $tap2$.

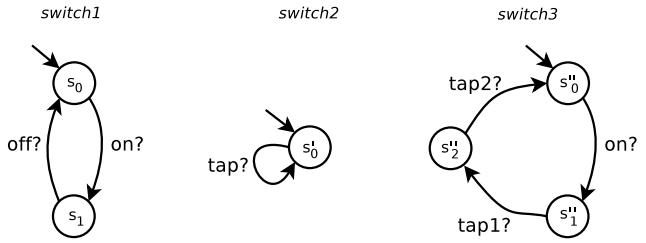


Figure 3. Example for Virtual Object based Substitution

Composition expression rewriting. When changing one object, the composition expression needs to be updated because it cannot interact anymore with object O but it must now interact with object O'. The solution proposed here aims at automatically updating the composition expression to take this change into account. This solution, as it was the case for the virtual object option, relies on a mapping between actions of the former and new device. Once we have this mapping we can replace actions in the rules of the composition expression. When there are several actions in a couple for a given object, we use the conjunction operator in the corresponding rule, because in that case, all actions need to be received or issued by the corresponding device.

Definition 7 (Composition Rewriting): Given an application defined by a set of objects O_i and a composition expression C, and given a new object O' replacing O ($O \in O_i$), C is rewritten based on the mapping M by substituting in C all occurrences of action in A by the corresponding actions in A' as defined in M.

Example. For illustration purposes, we take one IoT application consisting of one switch and one light (Fig. 4). According to R1, when the switch is pressed on, the light is turned on. R2 is used for turning off the light. The composition expression indicates that R1 and R2 apply infinitely in sequence. In the new application, we replace the switch by a more modern one where tapping is enough for alternatively switching on and off. The mapping is as follows (we will see how it is computed in the next section): $\{(on, tap), (off, tap)\}$. Therefore, by using this mapping, we can replace on and off by tap in rules R1 and R2, respectively, as shown on the right hand side of Figure 4.

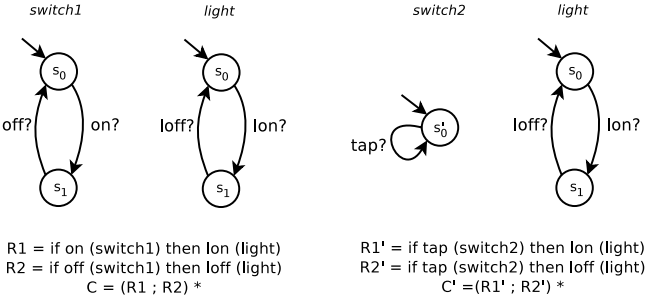


Figure 4. Example for Composition Rewriting based Substitution

Methodology. To conclude this (sub)section, let us introduce the simple methodology we suggest for the user who wants to use these substitution techniques. First, (s)he should try the simulation check. If the relation is verified, this is fine, and the deployment can be initiated (see Section IV). If simulation returns false, there are two options: or the user can choose another object or (s)he can try the other solutions (virtual object or composition rewriting). Note that these other solutions may not work either because the mapping step may fail (more details in the next subsection). However, if the mapping computation succeeds, these two solutions are equivalent in terms of results, that is, the application with the new object will exhibit at least the same functional behaviour as the application before substitution. Last but not least, if the user wants to change not one object but several ones, (s)he can apply our method several times in sequence.

B. Mapping Computation

The mapping computation works in several steps. Given the behavioural models (LTSs) of two objects (former and new object), we first compute the similarity matrix between the two LTSs (using the DLTS tool [9]). Then, we use this matrix to build couples of states with the highest similarity value in the matrix. Finally, for each couple, we look at outgoing transitions and actions hold on those transitions, and we use this information to build the mapping of actions. In the rest of this section, we will explain these three steps with more details, and we will illustrate them with an example.

Similarity measure. The first step relies on a comparison of both LTSs using a similarity measure [9] relying on bisimulation. This check first identifies bisimilar and non-bisimilar states. As for bisimilar states, since they are identical from a behavioural point of view, the mapping can be obtained in a straighforward way. Regarding non-bisimilar states, we need to go deeper in the comparison. For each couple of non-bisimular states (one non-bisimilar state from each LTS), we compute a degree of similarity which belongs to [0..1]. This value is computed using several local (e.g., comparison of actions on incoming and outgoing transitions) and global

criteria (position of similar states in their respective LTSs). All these results are stored in a matrix where non-bisimilar states of one LTS appear in row and non-bisimilar states of the other LTS appear in column.

Example. Figure 5 shows two versions of a motion sensor. The first one plays a sound when it detects a motion. The new one switches on a light for a few seconds instead. It is worth noting that there are exclamation marks at the end of labels in that case, because these events are raised by the device itself. The similarity measure computation returns the matrix given in Table I where we can see that all states are non-bisimilar because they all appear in the resulting matrix. For each state, there is a value indicating how similar this state is compared to any other state in the second LTS. As an example, states (s_0, s'_0) exhibit a similarity value of 0.71.

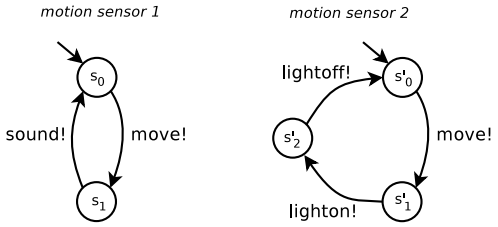


Figure 5. Example of Motion Sensors

	s'_0	s'_1	s'_2
s0	0.71	0.04	0.07
s1	0.02	0.68	0.38

Table I
EXAMPLE FOR THE LTS SIMILARITY MATRIX

State matching. The second step of the mapping construction aims at computing matches between the states of both LTSs. Note that, at this step, we do not have to match all states, but only those with clear matching. More formally, for each state in the first LTS, we look for the state in the second LTS with the highest similarity value. In case of bisimilar states, this is straightforward and a couple with the two states is added to the matching. In case of non-bisimilar states, we need to look at the similarity matrix. If the value computed for these two states is greater than a threshold (0.5 in our experiments), we add a couple of states to the matching. If there is no value greater than the threshold in the matrix for a given state, we do not match it with any other state. We will see in the final step that we do not require a match for all states. It is important to emphasize that, by relying on quantitative measures, this step does not systematically provide accurate results. Imagine for instance that the best score for a given state is 0.49. In such a case, we do not build any match for that state whereas there was perhaps one. Note that we consider that the resulting mapping is too imprecise if more than half of all states do not have any match. In this case, the mapping computation fails.

Example. If we go back to the example presented beforehand in this section, the state matching step provides the following result for the matrix given in Table I: $\{(s_0, s'_0), (s_1, s'_1)\}$. State s'_2 has no match since its highest value (0.38) is below the threshold.

Mapping computation. This final step takes as input the state matching obtained previously as well as the two input LTSs, and returns the mapping. There are several cases that we detail in Table II. In this table, the state matching step returns (s_0, s'_0) as one possible match and we show how we compute the mapping for that specific match. We can see in the first colum that we go through three possible cases, namely a single action, a choice between two actions and a sequence of two actions for which the intermediate state (s_1) has no match in the second LTS. In the second column, we see that we can have these three possible patterns as well. The whole table goes through all possible combinations.

Let us now comment on the three first cases, that will help the reader to understand the whole table. When there is one action on each side, we generate one couple with single actions, (a, b) in the table. If there is one action in the first LTS and a choice in the second LTS, we generate two couples with single actions (a, b) and (a, c), keeping in the mapping the non-determinism existing in the second LTS for that state. If there is a single action in the first LTS and a sequence of actions from s'_0 in the second LTS (with no state matching for s'_1), we generate an entry $(a, \{b, c\})$ in the mapping.

Note that Table II is not exhaustive. We may have other cases, such as one state with more than two outgoing transitions or one state with one outgoing transition and then a choice (with an intermediate state with no match). The mapping can be deduced for all these cases from the basic cases given in Table II.

Example. We go back one more time to the motion sensor example (Fig. 5). There are two matches in the state matching obtained during the second step. The first one (s_0, s'_0) exhibits a situation where each state has a single outgoing transition with a same label, so the first part of the mapping is straighforward, namely $(move, move)$. As for the second match (s_1, s'_1) (we recall there was no match for s'_2), we map one action with two actions, that is, $(sound, \{lighton, lightoff\})$.

IV. SUBSTITUTION IN ACTION

In this section, we present the substitution of IoT object in practice. This section consists of several parts. In a first part, we present how substitution is computed from a tool support perspective. In a second part, we show how IoT applications are executed with the Mozilla WebThings (WT) platform and how our substitution techniques and tools were integrated to that platform. Finally, we comment on some experiments we have carried out for validating the approach.

A. Tool Support for Substitution Computation

Figure 6 overviews the tools supporting the computation of our solutions for object substitution. Note that some steps rely on existing tools (CADP [10], DLTS [9]) and other steps are automated by tools we implemented for this work in Python.

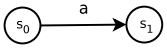
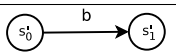
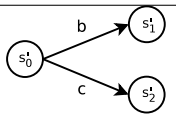
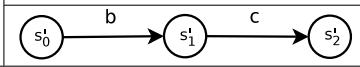
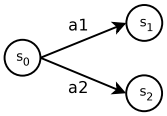
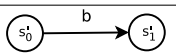
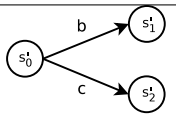
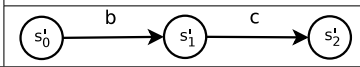
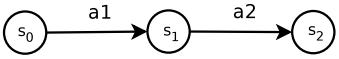
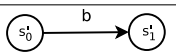
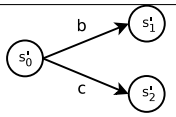
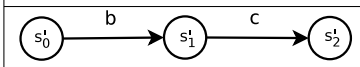
LTS1	LTS2	Mapping
		(a,b)
		(a,b), (a,c)
		(a,{b,c})
		(a1,b), (a2,b)
		(a1,b), (a1,c), (a2,b), (a2,c)
		(a1,{b,c}), (a2,{b,c})
		({a1,a2},b)
		({a1,a2},b), ({a1,a2},c)
		({a1,a2},{b,c})

Table II
MAPPING GENERATION FOR STATE MATCH (s_0, s'_0)

Let us now comment on this picture to explain all the steps of our tool support. First of all, the tool support takes as input an IoT application composed of a set of objects O_i and a composition expression C. We also need the new object O' and the object O to be replaced in the current application. Simulation is computed using the CADP toolbox, which is a rich verification toolbox implementing several analysis techniques for automata-based models. One of these techniques is equivalence checking, and can be used to check several notions of (bi)simulations and equivalences. In this work, we compare the two LTSs O and O' with respect to strong simulation ($\sim_s$) and we get a boolean as result. If the answer is true, it means that the new object simulates the former object and replacement is possible. If the answer is false, the simulation relation is not satisfied, and the user should go for another solution, by either choosing another new object or trying another option (virtual object or composition rewriting).

As for mapping generation, we recall that there are three steps as shown in Figure 6. The first step takes both objects O and O' as input and calls DLTS for computing the similarity matrix. The second step takes the matrix as input and computes a set of state matches. The third and final step takes the state matching as input and computes the mapping. Once the mapping is computed, we can use it (as well as the new object) for generating a virtual object in Thing Description (TD). As for composition rewriting, we start from the existing composition expression and we rewrite it using the mapping. All these steps (state matching, TD generation, composition expression rewriting) are automated by several Python scripts we implemented for this work.

B. Deployment and Substitution with WT

The IoT ecosystem is a diverse field consisting of various manufacturers with different underlying technologies and standards. Web of Things (WoT) is one of the standardization efforts to simplify building of IoT applications. It is based on the architectural styles and principles of the web, which is prevalent and thereby eliminates the need to learn various disparate technologies to build the applications. In WoT, objects are identified via a URI and each object has an associated Thing Description (TD), described in machine-readable JSON-TD. A TD of an object describes its behaviour, the operations it supports, i.e., the interfaces to monitor or alter its state, security configuration, and protocol bindings. The WoT standardization is led by W3C and at the time of writing is in Recommendation phase.

Mozilla WebThings [4] is a platform for monitoring and controlling devices over the web. It is based on Mozilla TD, a specification complementary to the W3C's work on abstract data model. In our work, we use the Mozilla specification as it provides a concrete implementation in the form of WebThings which can be extended quickly and efficiently. The Things UI component in WebThings allows users to build IoT automation in the form of "*If event(s) then action(s)*" Event-Condition-Action (ECA) rules. It also provides web APIs for monitoring

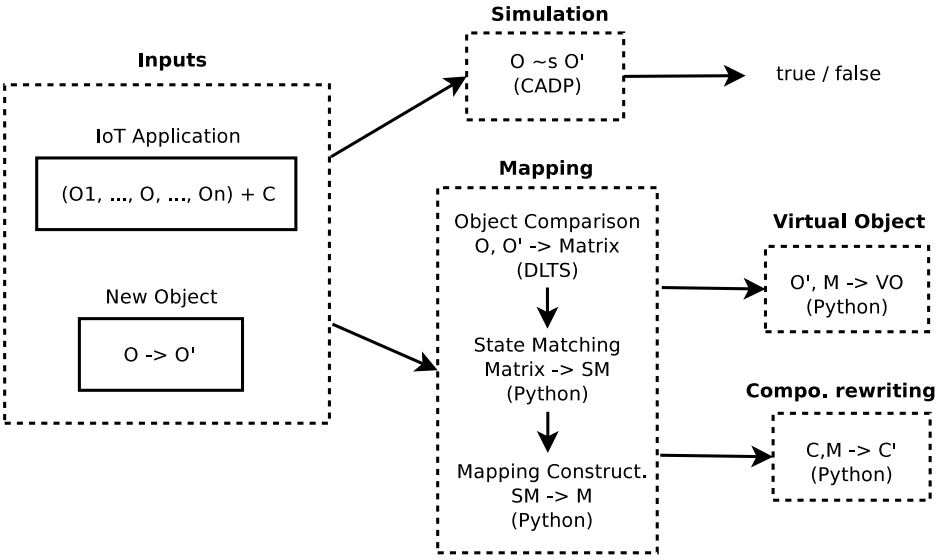


Figure 6. Substitution Tool Support Overview

and controlling IoT objects. Many of the popular objects are already supported by the platform and more objects are being constantly added.

Mozart [11] is a tool built on top of Mozilla WebThings to support the design and deployment of complex applications. In addition to individual ECA rules, it allows users to compose these rules using the composition language described in Section II. The tool supports the deployment of these applications. In the rest of this section, we explain how we have integrated the substitution techniques presented in this paper to Mozilla WebThings and Mozart in order to validate our approach in practice on smart home applications.

Given a running application and a new object replacing another one in the application, the substitution process applies successively the following tasks: (i) check for consistent substitutions using the tool support presented in Section IV-A, (ii) if step (i) is satisfactory, pause the application, make necessary updates, and restart the application.

Regarding step (i), the tool support was presented previously in this section. The additional need here is at the model transformation level. We have developed a converter transforming the IoT application described in the JSON format of the WebThings/Mozart platform to the input formats of the analysis tools (CADP, DLTS, our Python implementation).

As for step (ii), once the user decides to effectively replace the object by the new one, there are several steps that have to be carried out by a substitution manager we implemented. These steps depend on the technique used for substitution. If simulation is used for substitution, in that case, the manager pauses the application, that is, stops executing rules and stores all incoming events. Then, the manager replaces in the composition expression all references to the former object by references to the new object. It is worth noting that we do not have to add or remove objects since all objects are available in the physical environment. Once this is done, the manager restarts the application.

When using a virtual object, the manager first pauses the application. The virtual object is generated using the mapping, and then has to be deployed properly to appear in the WebThings list of available objects. The composition expression is updated as before by now referring to the virtual object instead of referring to the former object. Finally, the manager restarts the application.

As for the composition rewriting technique for substitution, beyond stopping/starting again the application as in the two other cases, the manager uses the mapping for rewriting the composition expression (as presented in section III). More precisely, rewriting replaces in the rules all events/actions of the former object by their counterparts in the mapping (new object), and therefore connects these updated rules to the new object.

As far as the user is concerned, here is what (s)he is supposed to do during the substitution process. First, (s)he has to choose an object to be replaced in the running application and a new object. Then (s)he can choose one substitution technique. If the substitution applies without any problem (e.g., simulation returns true), (s)he is asked to initiate the replacement and the substitution manager executes all the steps automatically for updating the application to integrate the new object. If there is a problem (e.g., failure of the state matching phase during the mapping generation or simulation returns false), the user receives a warning. In that case, (s)he should choose another new object. This situation happens when the new object is too different with respect to the former object, and the substitution does not make any sense.

C. Experiments

We have carried out a series of experiments on our own testbed simulating simple smart home applications in order to

validate our approach. This testbed consists of several objects including lights, motion sensors, screens and speakers. We used a control access system as scenario, which counts the number of people in a room (could be a shop for instance) and allows another person to access this room only if the max number of people has not been reached yet. Then, we made several versions of that application, by updating it step by step. At each step, we replace one object by another one (e.g., new motion sensor, colored light instead of basic light, speaker with voice messages replacing a speaker with only predefined sounds, etc.). For any replacement, we made use of our tool support for verifying whether the replacement preserved the application behaviour. We systematically tried the different options presented in Section III and finally deployed the new object using the substitution manager. These experiments allowed us to show that the two main steps of our approach (substitution verification and deployment) worked smoothly as expected.

As far as performance is concerned, all computations (simulation check, mapping computation, virtual object generation or composition rewriting) are very efficient ($<$1ms) because input models are usually small (less than 10 objects). Therefore, the substitution process as a whole only takes a few seconds corresponding to the time taken by the user for making his/her decisions (choice of objects and effective replacement decision).

V. Related Work

In [12], the author addresses the substitutability of component protocols described with Petri nets. The substitutability notion used in this paper is based on strong bisimulation. Replacing components using this relation enables to preserve system compatibility. In [13], the authors check component substitutability using weak bisimulation. They show that whenever there is a system in which a component is replaced with an observationally equivalent one, the system remains equivalent to the former one. This relation is less restrictive than strong bisimulation used in [12]. In [14], the authors use a Finite State Machine (FSM) model to formalise a substitutability notion for Web services which preserves compatibility. The authors consider a symmetric approach which requires that services must have the same traces. Pre/post-conditions are used and compared using a subtyping relation: the pre-conditions of an old service must be simulated by those of the new service and the post-conditions of the new service must be simulated by those of the old service. In [15], [16], service models are described using Labelled Transition Systems with a specific interest on internal behaviours. This work presents several notions of compatibility and replacement. Compared to these works, we go beyond strong notions of replacement such as bisimulation still preserving the functional behaviour of the whole application.

Substitution was also tackled in the context of software reconfiguration. Several formal models such as Darwin [17] or Wright [18] were proposed in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve (addition, removal or replacement of components and connections) at runtime. These techniques aim at helping users to formally design dynamic applications. In [19], [20], the authors show how to formally describe behavioural models of components using the FSP specification language and analyse these models using the Labelled Transition System Analyser (LTSA), which allows the verification of temporal properties on the component architecture. [21] presents a flexible approach to seamless reconfiguration of EJB-based Entreprise Applications. This work provides generic and reusable procedures for automatically supporting reconfiguration tasks. The role of the administrator is reduced to selecting an appropriate strategy and creating a reconfiguration plan that configures a generic procedure for a concrete reconfiguration. In this work, our goal was to maintain the global behaviour when substituting one object in an IoT application.

We are not aware of many works on substitution of device in the IoT area. [22] introduces the OpenPnP reference architecture, which allows a significant reduction of configuration and integration effort during industrial plant commissioning. The OpenPnP architecture reduces configuration and installation time by up to 90 percent, while scaling to Industrial IoT systems with many nodes. OpenPnP also provides concepts for replacing malfunctioning devices. This work does not attempt to preserve any guarantee regarding the behaviour of the application before and after substitution. In [23], [24], the authors propose new techniques for supporting the reconfiguration of running IoT applications. These techniques compare two versions of the application (before and after reconfiguration) to check if several properties of interest from a reconfiguration perspective are preserved. The analysis techniques have been implemented using the Maude framework and integrated into the WebThings platform. In this work, we focus on the replacement of a single object and we aim at providing formal guarantees in this specific context.

VI. Concluding Remarks

We have presented in this paper a set of techniques for supporting the substitution of one object in a running IoT application. In this work, we assumed that substitution should not change the whole behaviour of the application from an observational point of view. This is particularly useful when one needs to replace one object due to a failure, maintenance, software update, new device, etc., but does not want to change the overall behaviour of the application. We proposed three different techniques for ensuring that substitution is consistent in that way. The first one relies on simulation checking, which compares the behavioural models of both objects. The second and third techniques first compute a mapping between the actions of both objects. Then, they exploit this mapping for either generating a virtual object simulating the behaviour of the former object on top of the new object, or for automatically rewriting the composition expression to use the new object instead of the old one. Our solutions are supported and automated by several tools we reused or have implemented ourselves. In particular, our contributions

on substitution analysis and computation were integrated to the Mozilla WebThings platform. This allowed us to make experiments on smart home applications, showing that our approach works correctly on real IoT applications.

As far as future work is concerned, we would like to make our approach fully automated. In its current form, human intervention is required for choosing an object to be replaced and a new object. At some point, in the substitution process, the user is also asked to initiate the effective replacement. A fully automated approach would be very useful, particularly in case of failure, in order to trigger the substitution process without expecting any manual intervention. Another perspective aims at providing additional notions of replacement such as deadlock-freeness like suggested in [25]. This would allow users to have other possible options. However, in that case, it may not be possible to preserve the very same behaviour of the application (a.k.a. application consistency).

REFERENCES

[1] S. Mittal, W. T. Tam, and C. Ko. (2018) Internet of Things: The pillar of artificial intelligence.

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

[3] O. Foundation. (2020) Node-red: Low-code programming for event-driven applications. [Online]. Available: https://nodered.org/

[4] Mozilla, "WebThings," 2020. [Online]. Available: https://iot.mozilla.org/

[5] IETF, "Constrained RESTful Environments," 2020. [Online]. Available: https://datatracker.ietf.org/wg/core/charter/

[6] Nest Labs, "A Secure and Reliable Communications Backbone for the Connected Home," 2020. [Online]. Available: https://openweave.io/

[7] W3C, "Web of Things at W3C," 2020. [Online]. Available: https://www.w3.org/WoT/

[8] R. Milner, *Communication and Concurrency*. Prentice Hall, 1989.

[9] G. Salaün, "Quantifying the Similarity of Non-bisimilar Labelled Transition Systems," *Sci. Comput. Program.*, vol. 202, 2021.

[10] 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.

[11] A. Krishna, M. L. Pallec, A. Martinez, R. Mateescu, and G. Salaün, "MOZART: Design and Deployment of Advanced IoT Applications," in *Proc. of WWW'20*. ACM, 2020.

[12] N. Hameurlain, "On Compatibility and Behavioural Substitutability of Component Protocols," in *Proc. of SEFM'05*. IEEE Computer Society, 2005, pp. 394–403.

[13] I. Cerná, P. Vareková, and B. Zimmerova, "Component Substitutability via Equivalencies of Component-Interaction Automata," *Electron. Notes Theor. Comput. Sci.*, vol. 182, pp. 39–55, 2007.

[14] H. S. Chae, J. Lee, and J. H. Bae, "An Approach to Checking Behavioral Compatibility between Web Services," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 18, no. 2, pp. 223–241, 2008.

[15] M. Ouederni and G. Salaün, "Tau Be or not Tau Be? - A Perspective on Service Compatibility and Substitutability," in *Proc. of WCSI'10*, ser. EPTCS, vol. 37, 2010, pp. 57–70.

[16] 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.

[17] J. Magee and J. Kramer, "Dynamic Structure in Software Architectures," in *Proc. of SIGSOFT FSE'96*, 1996, pp. 3–14.

[18] R. Allen, R. Douence, and D. Garlan, "Specifying and Analyzing Dynamic Software Architectures," in *Proc. of FASE'98*, ser. LNCS, vol. 1382. Springer, 1998, pp. 21–37.

[19] J. Kramer and J. Magee, "Analysing Dynamic Change in Distributed Software Architectures," *IEE Proceedings - Software*, vol. 145, no. 5, pp. 146–154, 1998.

[20] J. Magee, J. Kramer, and D. Giannakopoulou, "Behaviour Analysis of Software Architectures," in *Proc. of WICSA'99*, ser. IFIP Conference Proceedings, vol. 140. Kluwer, 1999, pp. 35–50.

[21] T. Vogel, J. Bruhn, and G. Wirtz, "Autonomous Reconfiguration Procedures for EJB-based Enterprise Applications," in *Proc. of SEKE'2008*. Knowledge Systems Institute Graduate School, 2008, pp. 48–53.

[22] H. Koziolek, A. Burger, M. Platenius-Mohr, J. Rückert, and G. Stomberg, "OpenPnP: A Plug-and-produce Architecture for the Industrial Internet of Things," in *Proc. of ICSE(SEIP)'19*. IEEE / ACM, 2019, pp. 131–140.

[23] F. Durán, A. Krishna, M. L. Pallec, R. Mateescu, and G. Salaün, "Seamless Reconfiguration of Rule-based IoT Applications," in *Proc. of SEAMS'21*. ACM, 2021.

[24] F. Duran, A. Krishna, M. L. Pallec, R. Mateescu, and G. Salaün, "R-MOZART: A Reconfiguration Tool for WebThings Applications," in *Proc. of ICSE'21*. IEEE / ACM, 2021.

[25] L. Bordeaux, G. Salaün, D. Berardi, and M. Mecella, "When are two web services compatible?" in *Proc. of TES'04*, ser. LNCS, vol. 3324. Springer, 2004, pp. 15–28.