Seamless Reconfiguration of Rule-based IoT Applications

Francisco Durán[*], Ajay Krishna[†], Michel Le Pallec[‡], Radu Mateescu[†] and Gwen Salaün[§]
[*]ITIS Software, University of Málaga, Spain
[†]Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG 38000 Grenoble, France
[‡]Nokia Bell Labs 91620 Nozay, France
[§]Univ. Grenoble Alpes, CNRS, Grenoble INP, Inria, LIG 38000 Grenoble, France

Abstract—**The Internet of Things (IoT) consists of devices and software interacting altogether in order to build powerful and added-value services. One of the main challenges in this context is to support end-users with simple, user-friendly, and automated techniques to design such applications. Given the dynamicity of IoT applications, these techniques should consider that these applications are in most cases not built once and for all. They can evolve over time and objects may be added or removed for several reasons (replacement, loss of connectivity, upgrade, failure, etc.). In this paper, we 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.**

I. Introduction

Designing IoT applications by selecting a set of candidate objects and defining how they interact with one another is a difficult and error-prone task. Indeed, several kinds of mismatch can arise, or the overall objective may not be fulfilled due to wrong design, resulting in an erroneous application. Moreover, 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. In this paper, we propose new techniques for supporting the reconfiguration of IoT applications. Updating an IoT application should be carried out with specific care because this may induce incorrect behaviour or inconsistencies in the application. Such problems must be avoided during the reconfiguration process because they can induce additional costs or hazardous situations. As an example, in the context of IoT applications for smart homes and buildings, simply switching off lights, heaters or closing doors unexpectedly upon reconfiguration may cause harm to humans.

This paper presents an approach for analysing the proposed reconfiguration to check if it preserves the consistency of the application, i.e., the application can resume after reconfiguration from where it was before interruption. More precisely, an IoT application is described in this work using a set of objects and a rule-based composition expression that specifies how the objects interact together. Given a current and a new IoT application as well as a global state of the current application, the main reconfiguration property (called *seamless* reconfiguration) determines if the given global state is reachable for objects remaining in the new application. If this is the case, it means that replacing the current application by the new application can be achieved seamlessly from the user perspective. We also define two additional properties called *conservative* reconfiguration and *impactful* reconfiguration to check whether all former behaviours can still be executed in the new application, and whether all newly introduced behaviours can be executed after reconfiguration, respectively. These three properties focus on the reconfiguration of the application taking into account its global state. Complementary to these properties, functional properties of interest can be verified on the new application only. These properties are analysed for all possible executions of the application independently of any global state.

As far as tool support is concerned, R-Mozart is presented in a companion paper [1]. It consists of three components for supporting the reconfiguration of IoT applications. At the design level, a new UI allows the user to specify the new application. Once the new application is specified, a verification component checks whether the reconfiguration properties are satisfied. To check these properties, we provide an encoding into the rewriting logic of the Maude framework [2], and we rely on Maude tools to generate and traverse all possible executions of both applications. Finally, deployment of the new configuration is achieved preserving the consistency of the remaining objects. Note that we take a human-in-the-loop approach, where reconfiguration is defined by users, and the steps of verification and deployment are automated through the components we implemented. This setup has been validated on several examples.

The rest of this paper is organised as follows. Section II introduces the model of objects and the rule-based composition language. Section III defines properties of interest for reconfiguration of ruled-based IoT applications. Section IV presents the encoding of IoT applications into rewriting logic and the verification of properties using Maude. Section V surveys related work and Section VI concludes the paper.

II. IoT Models

An IoT application consists of a set of IoT objects or *things* interacting all together to fulfil a certain overall goal.

In this work, we rely on an abstract model of objects. An IoT object is defined by a set of properties (a property is a pair (*identifier*, *value*)) and by a behavioural model. For the sake of simplicity, an IoT object is represented only by its behavioural model in the rest of this paper. The objects can indicate and change their states using events and actions, respectively. The behavioural model, defined in terms of a Labelled Transition System (LTS), represents the ordering of events/actions in an object. A question mark (?) and an exclamation mark (!) are used to indicate that the object is receiving or emitting a value from/to its environment, respectively.

Definition 1 (IoT Object): An IoT Object O is modelled as a Labelled Transition System $LTS = (S, A, T, s^0)$, 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, which acts like an orchestrator indicating how the involved objects interact together. A simple rule-based composition language is used to specify the expression. It 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 by its object identifier.

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

EVT ::= *event* (Oid) | EVT$_1$ and EVT$_2$ | EVT$_1$ or EVT$_2$,
ACT ::= *action* (Oid) | ACT$_1$ and ACT$_2$,

where the terminal symbols are $event, action \in \bigcup_{i=1}^{n} A_i$, and $Oid \in \{1, \dots, n\}$ 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. Note that the composition language is a regular language. Thus, any composition expression C written with this language can be transformed into an LTS with rules as labels. In the rest of this paper, we use interchangeably the terms composition expression and composition LTS.

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

$$C ::= R \mid C_1 \; ; \; C_2 \mid C_1 + C_2 \mid C_1 \parallel C_2 \mid C_1^k$$

where $C_1 \; ; \; C_2$ represents a composition followed by another composition, $C_1 + C_2$ represents a choice between two composition expressions, $C_1 \parallel C_2$ represents the concurrent execution of a two composition expressions, and C_1^k represents the execution k times of a composition expression (if $k = *$, C_1 executes a finite, unspecified number of times).

Let us now explain how an IoT application, consisting of a set of objects and a composition expression, executes. The communication model being asynchronous, each object is equipped with an input message buffer (FIFO). The composition expression and all objects start their execution from their initial states. Then, an application can evolve in two cases: execution of a rule or buffer consumption. In the first case, let us assume a basic rule with one event and one action. If the event appearing in the left part of the rule has been issued, the rule can be triggered and the action appearing in the right part of the rule is pushed to the corresponding object's buffer. The event 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 toggling 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 consume according to its LTS model. In both bases, the global state of the application changes. A global state consists of the current state of all objects involved in the application and the current state of the composition expression/LTS.

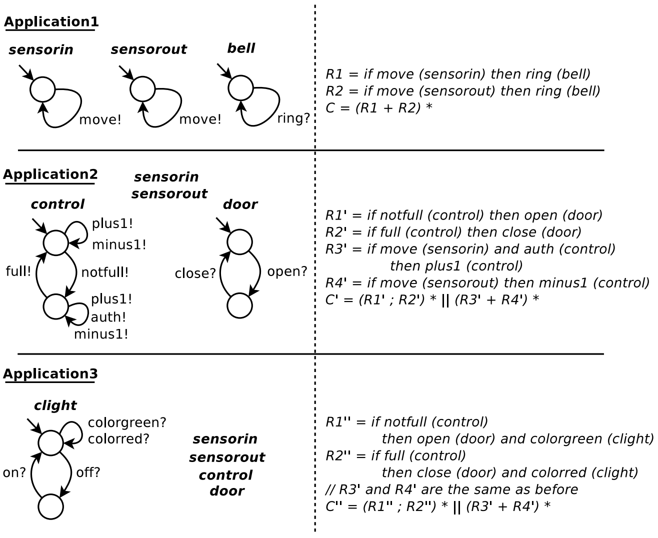
Definition 4 (One-step Execution Semantics): Given an IoT application $(\{O_1, \dots, O_n\}, C)$ defined by a set of objects $O_i = (S_i, A_i, T_i, s_i^0)$, $i = 1, \dots, n$ (with B_i the input buffer for object O_i) and by a composition LTS $C = (S, A, T, s^0)$, and given its current global state $(((s_1, B_1), \dots, (s_n, B_n)), s)$, the application can evolve as follows:

- (rule execution)
$(((s_1, B_1), \dots, (s_j, B_j), \dots, (s_k, B_k), \dots, (s_n, B_n)), s) \xrightarrow{m!}$
$(((s_1, B_1), \dots, (s'_j, B_j), \dots, (s_k, B_k m'), \dots, (s_n, B_n)), s')$
if $\exists j, k \in \{1, \dots, n\}$, $j \neq k$, $s_j \xrightarrow{m!} s'_j \in T_j$, and $s \xrightarrow{if \; m \; (j) \; then \; m' \; (k)} s' \in T$.
- (buffer consumption)
$(((s_1, B_1), \dots, (s_j, mB_j), \dots, (s_n, B_n)), s) \xrightarrow{m?}$
$(((s_1, B_1), \dots, (s'_j, B_j), \dots, (s_n, B_n)), s)$
if $\exists j \in \{1, \dots, n\}$, $s_j \xrightarrow{m?} s'_j \in T_j$.

At a given global state, several executions may be possible due to the use of choice and interleaving operators in the composition expression. By applying the one-step execution semantics whenever it is possible, we can cover all executions of the IoT application and thus give an LTS-based semantics to such applications. In the former definition, we use a basic rule for simplicity. It could easily be extended to all kinds of rules presented in Def. 2. In the rest of this paper, we need to guide the execution of an IoT application by a given trace. A trace is a sequence of couples *(object identifier, action)*, where an action is either an occurrence of the event associated with a rule or a buffer consumption as shown in the former definition.

Definition 5 (Trace): Given an application $\mathcal{A} = ((O_1 \dots O_n), (S, A, T, s^0))$, a trace t is an ordered sequence of pairs $< (oid_1, a_1), \dots, (oid_m, a_m) >$, where, for each pair, oid is the identifier of an object O_i, and a is either an event $m!$ that triggers a rule or an action $m?$ consumed from a buffer.

Assuming that the LTSs of the objects and of the composition expression are deterministic, the execution is deterministic and it is defined by the actions appearing in that trace. In our work, we also use the notion of filtered trace, which consists



```
R1 = if move (sensorin) then ring (bell)
R2 = if move (sensorout) then ring (bell)
C = (R1 + R2) *

R1' = if notfull (control) then open (door)
R2' = if full (control) then close (door)
R3' = if move (sensorin) and auth (control)
        then plus1 (control)
R4' = if move (sensorout) then minus1 (control)
C' = (R1' ; R2') * || (R3' + R4') *

R1'' = if notfull (control)
        then open (door) and colorgreen (clight)
R2'' = if full (control)
        then close (door) and colorred (clight)
// R3' and R4' are the same as before
C'' = (R1'' ; R2'') * || (R3' + R4') *
```

Figure 1. Several versions of a shop access control

of selected pairs from the original trace, belonging to the remaining objects in a reconfigured application.

Definition 6 (Filtered Trace): Given an application $\mathcal{A}_{curr}$ consisting of objects O_{curr}, its reconfigured version $\mathcal{A}_{new}$ consisting of objects O_{new}, and a trace t corresponding to the sequence of actions executed in $\mathcal{A}_{curr}$, the filtered trace t_f consists of pairs from t appearing in the same order, and for each pair $t_{elem} = (oid, a)$ of t, t_{elem} also belongs to t_f iff oid is the identifier of an object $O_i \in O_{curr} \cap O_{new}$.

Example. Application1 given in Figure 1 consists of two motion sensors and a bell. This application can be used to notify the entry and exit of customers in a shop. The composition expression (right hand side in Figure 1) consists of two rules: when the motion sensor detects someone entering the shop it rings the bell, and when the second motion sensor detects someone leaving the shop it again rings the bell. The composition can execute these rules at any time by using a choice (+) and an iteration construct (*). Application2 and Application3 in Figure 1 are the reconfigured versions of Application1 and Application2, respectively.

III. Reconfiguration Properties

In this section, we present several properties that can be checked before replacing the current application by a new application. First, we present three properties that take as input two applications, the current and the new one, as well as the global state of the current application before reconfiguration. These properties are called *seamless*, *conservative*, and *impactful* reconfiguration, and assess the impact of replacing the current application by the new one in its current global state. At the end of the section, we show how other properties of interest could be verified on the new application without starting from a specific global state, but considering all possible executions of this new application.

A. Seamless Reconfiguration

To check the *seamless* reconfiguration property, we need the following inputs: the current application, a new application (both defined by their respective set of objects and composition expressions), and the current global state of the current application (i.e., the application before reconfiguration). The seamless reconfiguration property checks whether this state can be reached again in the new application. This is important because it means that the deployment of the new application is possible without starting again this application from the beginning, thus seamlessly replacing the current application by the new application from the perspective of the user.

When reconfiguring an application, one can remove objects, add new objects, and change the composition expression. In this work, we consider that an application can be seamlessly reconfigured if all the remaining objects (i.e., common to the current and new applications) can reach again the state where they were before initiating the reconfiguration, according to the new composition expression. We focus on remaining objects because the states of removed objects do not need to be restored and new objects can start their behaviour from any state (they do not have an execution history). Since each remaining object must reach again the state where it was before reconfiguration, we also need the trace executed by the current application from its initial state up to the current global state. This trace is useful to check whether there is one execution of the new composition expression where all remaining objects can reach their former states repeating the same behaviour. This trace is obtained by instrumenting the IoT platform (WoT in this work) to capture all the events/actions issued by the objects involved in the application.

When simulating the new application execution, guided by a trace which was executed on the current application, the remaining objects have to repeat the same actions. As far as the new objects are concerned, they evolve with respect to the new composition expression whose evolution is guided by the trace. The states in which the new objects would have to start on deploying the new application are obtained by executing the trace on the new application.

Definition 7 (Seamless Reconfiguration): Given two applications $\mathcal{A}_{curr} = (O_{curr}, C_{curr})$ and $\mathcal{A}_{new} = (O_{new}, C_{new})$, each defined by a set of objects and a composition expression, given the current global state $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ of application $\mathcal{A}_{curr}$ and the trace t to reach that state consisting of a sequence of couples (object identifier, action), the seamless reconfiguration property is satisfied if, when executing application $\mathcal{A}_{new}$ guided by the trace t, each remaining object $O_i \in O_{curr} \cap O_{new}$ starting from s_i^0 can execute the actions in t and reach its current state s_i.

Example. Let us consider again the simple shop application used previously in this paper (Figure 1). Suppose we want to replace Application1 by Application2. In Application2, the entrance is controlled in order to limit the number of people in the shop (e.g., to 10 people). This situation has become quite usual in many places since the Coronavirus outbreak.

To do so, we keep both sensors, and add an access control and a door. The access control is a software-based counter that counts the presence of customers, detects when the shop is full, and accordingly authorises or restricts the entry of new customers. The rules are configured so that the entry door is kept open when the number of customers inside is below a certain threshold value and the door is closed when the shop is full. Replacing the first application by the second one corresponds to a seamless reconfiguration because the two remaining objects (sensorin and sensorout) have a single state, which is reachable in the new composition expression for any execution trace.

Suppose now that we want to add a coloured light to Application2, thus becoming Application3 (Figure 1). When the access is possible to the shop, the door is opened and the light is green, whereas when the access is prohibited, the door is closed and the light is red. This reconfiguration (from Application2 to Application3) is seamless as well, irrespective of the previous execution. Indeed, all remaining objects (sensorin, sensorout, control and door) can repeat their former behaviours since the composition expression for Application3 is an extension of the composition expression for Application2, so all former behaviours are still possible in the new application. Notice that it is important that the seamless property is satisfied because we want to keep the application in a consistent state during the reconfiguration process, i.e., maintain the state of the door either open or closed depending on the number of customers in the shop.

B. Conservative and Impactful Reconfigurations

The seamless reconfiguration definition indicates whether the remaining objects can reach again their former states in the new application. We can go further than this initial check by comparing more precisely both applications in terms of preserved behaviours and new behaviours. Therefore, we propose a couple of additional properties that could be helpful in order to better characterise the intended reconfiguration before applying it in practice.

A reconfiguration is called *conservative* if the seamless property is preserved and if, from the global state in which the reconfiguration is applied, all behaviours that could be executed in the current application (objects and composition expression) are still executable in the new application. Thus, everything that was possible before is still possible in the new application from that state (each trace that can be executed in the current application is still executable in the new one). This check is useful when one wants an application to provide more services or features still preserving exactly what was possible before. Note that the global state of the current application cannot be used as a starting point in the new application because some objects may have been removed or added. To obtain the "equivalent" global state in the new application, we use the trace executed by the current application in the new application: remaining objects replay the same events/actions and new objects evolve following the new composition expression. In this way, we are able to compute a global state in the new application (the one computed to check that the seamless property is satisfied), and we use that state as starting point for checking conservative reconfiguration.

Definition 8 (Conservative Reconfiguration): Given two applications $\mathcal{A}_{curr} = (O_{curr}, C_{curr})$ and $\mathcal{A}_{new} = (O_{new}, C_{new})$, given the current global state $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ of application $\mathcal{A}_{curr}$ and the trace t that was executed to reach that global state, the conservative reconfiguration property is satisfied if the seamless reconfiguration property is satisfied and if each trace t' that can be executed in $\mathcal{A}_{curr}$ from $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ can also be executed (after filtering it on the remaining objects $O_{curr} \cap O_{new}$) in $\mathcal{A}_{new}$ from $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$ where the global state $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$ is obtained by executing $\mathcal{A}_{new}$ guided by t.

A reconfiguration is called *impactful* if the seamless property is preserved and if the whole behaviour of each new object can be entirely executed in the new application. To check that, we first execute the former trace to obtain the global state in the new application. Then, we compute all behaviours that are reachable from that global state according to the new composition expression. The entire behaviour of each new object must be covered to say that the reconfiguration is impactful. This property allows one to verify whether the newly introduced behaviours are fully utilised in the new application.

Definition 9 (Impactful Reconfiguration): Given two applications $\mathcal{A}_{curr} = (O_{curr}, C_{curr})$ and $\mathcal{A}_{new} = (O_{new}, C_{new})$, given the current global state $(((s_1, B_1), \ldots, (s_n, B_n)), s)$ of application $\mathcal{A}_{curr}$ and the trace t that was executed to reach that state, the impactful reconfiguration property is satisfied if the seamless reconfiguration property is satisfied and if each new object $O_i \in O_{new} \backslash O_{curr}$ has its entire behaviour appearing in $\{t\} \cup Tr$ (i.e., for each O_i, for each $s_1 \xrightarrow{ed} s_2 \in T_i$, e appears at least once in $\{t\} \cup Tr$), where $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$ is the global state obtained by executing $\mathcal{A}_{new}$ guided by t and Tr is the set of all traces that can be executed in A_{new} from $(((s'_1, B'_1), \ldots, (s'_m, B'_m)), s')$.

Note that conservative and impactful reconfiguration properties are independent of each other. If all the objects and the composition expression are in their respective initial states, the conservative property is not systematically preserved, but the impactful property may be not preserved either because the new composition expression may prevent some new behaviour to be executed.

Example. Let us focus first on the replacement of Application1 by Application2 (Figure 1). For simplification purposes, we assume that the execution trace makes all LTSs (for objects and composition expression) come back to their initial states, which is thus considered as the starting global state for analysing these properties. This reconfiguration is not conservative as the bell has been removed and this behaviour (ring) cannot be executed any more. This reconfiguration is impactful because the entire behaviour of the new objects (control and door) can be executed according to the new composition expression C'. As far as the reconfiguration from Application2 to Application3 is concerned, we assume again that the reconfiguration takes place when all the object and composition LTSs are in their initial states for simplicity. This reconfiguration is conservative because the behaviour that can be executed in the current application can be executed in the new application too. This reconfiguration is not impactful because a part of the new object (clight) is not reachable with respect to the composition expression C'' as there are no rules executing the actions on and off.

C. Additional Properties

Beyond the reconfiguration properties introduced previously in this section, it is also possible to check classic safety and liveness properties on the new application (only) using model checking techniques. In this case, this additional verification does not focus on a specific global state, but it allows the analysis of all the possible executions of the new application. Deadlock freeness for instance can be checked on the new application. This property is generic in the sense that it does not depend on the application. In contrast, other properties may depend on the application. For instance, if we go back to our example (Application3), we could verify that the door is eventually closed (when the number of customers is more than the threshold).

IV. Specification and Verification in Maude

This section shows how the different properties presented in the former section can be automatically checked via an encoding into rewriting logic and the use of Maude's verification tools [2]. Reconfiguration analysis mainly involves simulation and traversal of execution paths of an IoT application that are easily expressed using equational logic and efficiently computed using term rewriting in Maude. The code is available online [3] with examples.

A. Formal Specification

Maude is a high-level language and a high-performance system that supports membership equational logic, and rewriting logic specification and programming of systems, see [2] for a detailed introduction of Maude. The implementation in Maude of an IoT application consists of four steps, which aim at specifying successively IoT objects or devices, rules, composition expressions, and applications. As stated in Section II, an object (Listing 1) is described by an LTS consisting of an initial state and a set of transitions (the alphabet and the set of states can be deduced from the set of transitions). A rule is defined as a single event or a set of events (and/or) in the left part, and as a single action or a set of actions in the right part. A composition (Listing 1) can make use of all the operators introduced in Section II (sequence, choice, parallel, iteration). Finally, an application consists of a set of objects and a composition expression.

```
fmod LTS is
   pr STATE .
   pr SET{Transition} .

   sort LTS .
   op model : State Set{Transition} -> LTS .
endfm

fmod DEVICE is
   pr LTS .

   sort Device .
   op dev : Id LTS -> Device .
endfm

fmod COMPOSITION is
   pr RULE .
   pr INT .

   sort Composition .
   subsort Rule < Composition .
   op seq : Composition Composition -> Composition [assoc
      ↪right id: none] .
   op ch : Composition Composition -> Composition [comm] .
   op par : Composition Composition -> Composition [comm] .
   op iter : Composition Int -> Composition .
   op none : -> Composition .
endfm
```

Listing 1. Definition of the composition language

B. Automated Analysis

The seamless reconfiguration property is encoded as an operation in Maude which takes as input the current application, the new application, the global state reached by the current application, and the trace executed by the current application to reach that state. It returns a Boolean response indicating whether the current global state for the remaining objects is reachable by executing that trace. New objects can be involved in order to reach that state, but whatever state they reach, it does not impact the seamless reconfiguration property.

This property is checked by first executing the trace in the new application and returning the reached global state. This state is unique because the LTS models of the new application are deterministic and the execution is guided by the given trace. Then, we check whether both global states coincide for the set of remaining objects. Note that the Maude specification precisely encodes the execution semantics of the models described in Section II. In particular, each object is equipped with an input buffer for modelling the communication model used in our IoT application model. Listing 2 shows a few operations used for computing the seamless reconfiguration property. The first operation (checkSeamlessReconfiguration) takes as input all required elements (two applications, one trace and one global state) and filters out all events/actions in the trace that do not belong to the set of remaining objects. The second operation (checkSeamlessReconfigurationAux) takes as input two applications, one global state, the trace (filtered to keep only actions executed by the remaining objects) and the set of remaining objects. This operation calls the auxiliary function runTrace to execute the second application guiding this execution by the trace. If an object is not available anymore (removed in the second application), any object can be run instead (yet according to the new composition expression). The

```
op checkSeamlessReconfiguration :
    Application Application Set{Tuple{Id,State}} List{Tuple{Id,Label}} -> Bool .
op checkSeamlessReconfigurationAux :
    Application Application Set{Tuple{Id,State}} List{Tuple{Id,Label}} Set{Id}
      -> Bool .

---- filters the trace to keep only labels belonging to remaining objects
eq checkSeamlessReconfiguration(App1, App2, GS, Tr)
  = checkSeamlessReconfigurationAux(
      App1, App2, GS,
      filterTrace(Tr, computeCommonObjects(App1, App2)),
      computeCommonObjects(App1, App2)) .

---- runs the trace in the new application until it is possible
eq checkSeamlessReconfigurationAux(App1, App2, GS, Tr, Ids)
  = compareGS(
      App1, App2, GS,
      getGlobalState(runTrace(App2, Tr, Ids)))
    and
    getBoolRes(runTrace(App2, Tr, Ids)) .
```

Listing 2. Specification of the seamless reconfiguration property

operation compareGS checks that the remaining objects have reached the same state in both global states. The operation runTrace also returns a Boolean value indicating whether the whole trace was executed for the remaining objects.

As far as the conservative property is concerned, we first check that seamless reconfiguration is preserved. Then, we start from the computed global state in the new application. We execute all possible behaviours in the new application, and we check that there is a match in the current application for each possible trace. We stop when we have traversed all behaviours and they all match, or when there is a mismatch.

The impactful property checks that all new behaviours can be executed in the new application. First, seamless reconfiguration is verified and we start from the global state returned by this initial check. Second, we focus only on the second application and we compute and store all observable events for each device (input and output) from that state following the new composition expression. Finally, we check that all events have been traversed, for new devices only, from their respective initial states.

The analysis of classic safety and liveness properties is achieved by using the object-oriented and rule-based capabilities of Maude. Given an application (a set of objects and a composition expression), we define a class called Simulation with four attributes: the current global state, the current trace, the current state of the composition expression, and a set of buffers (one input buffer per object). Then, we define six rules corresponding to all possible evolutions of our system. There are five rules corresponding to the evolution of the composition expression (one rule for sequence, choice, interleaving, and two rules for iteration), and one rule corresponding to the consumption by one object from its buffer.

To conclude this section, let us comment on the performance of the reconfiguration analysis with Maude. Our experiments show that it takes only a few milliseconds to check the reconfiguration properties introduced in this paper (seamless, conservative or impactful) on simple examples (less than 10 objects). This is because the check just compares the traces from the global state. However, a comprehensive traversal of the behaviour is required for checking classic safety and liveness properties.

V. Related Work

Dynamic reconfiguration is one of the main problems in software architectures, where several formal frameworks such as Darwin [4] or Wright [5] were proposed in order to specify dynamic reconfiguration of component-based systems whose architectures can evolve at runtime (by adding or removing components and connections). Seamless reconfiguration is not a new notion and was used in several works focusing on dynamic reconfiguration, e.g., [6], [7]. As an example, [7] presents a flexible approach to seamless reconfiguration of EJB-based enterprise applications. As far as reconfiguration of component assemblies is concerned, the authors present in [8], [9] a reconfiguration protocol applying changes to a set of connected components for transforming a current assembly to a target one given as input. In [10], the authors propose a reconfiguration protocol for dynamically updating a cloud application consisting of components deployed on virtual machines. [11], [12] present modelling and verification techniques for supporting the design of IoT applications. The approach presented in [13] extends semantic application descriptions (called recipes) with constraints to enable dynamic and automatic reconfiguration of IoT applications. Using recipes, dynamic choreographies can be created that self-adapt to changing device states without human intervention. [14] introduces the OpenPnP reference architecture, which allows a significant reduction of configuration and integration efforts during industrial plant commissioning. Our focus here is not only on the design of IoT applications but also on their analysis in order to assess the impact of reconfiguration on the application from a consistency and correctness perspective.

VI. Conclusion

In this paper, we have focused on IoT applications consisting of devices interacting as described in a composition expression of rules. These applications are not built once and for all any more. This work gives the possibility to change these applications (addition or removal of objects, update of the composition expression) and provides formal guarantees during the reconfiguration process. We have defined several properties that characterise the consistency and correctness of the application to be reconfigured. We have also proposed verification techniques that allow one to analyse not only the update of an application with respect to a certain global state of the application, but also to analyse all possible executions of the new application to check whether it preserves certain functional properties. All these checks are fully automated using an encoding into rewriting logic, and Maude's simulation and model checking tools. R-Mozart [1] implements the ideas presented in this paper and thus supports the design, verification and deployment of a new IoT application.

References

[1] F. Duran, A. Krishna, M. L. Pallec, R. Mateescu, and G. Salaün, "R-MOZART: A Reconfiguration Tool for WebThings Applications," in *Proc. of the 43rd International Conference on Software Engineering: Companion Proceedings, ICSE 2021, Virtual Event*. IEEE / ACM, 2021.

[2] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. L. Talcott, Eds., *All About Maude - A High-Performance Logical Framework, How to Specify, Program and Verify Systems in Rewriting Logic*, ser. LNCS. Springer, 2007, vol. 4350.

[3] F. Durán, A. Krishna, M. L. Pallec, R. Mateescu, and G. Salaün, "Seamless Reconfiguration of Rule-based IoT Applications," https://github.com/ajaykrishna/rmozart/tree/main/maude, January 2021.

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

[5] R. Allen, R. Douence, and D. Garlan, "Specifying and analyzing dynamic software architectures," in *Proceedings of the Fundamental Approaches to Software Engineering, FASE'98*, ser. Lecture Notes in Computer Science, E. Astesiano, Ed., vol. 1382. Springer, 1998, pp. 21–37. [Online]. Available: https://doi.org/10.1007/BFb0053581

[6] L. Rosa, L. E. T. Rodrigues, and A. Lopes, "A framework to support multiple reconfiguration strategies," in *Proceedings of the 1st International Conference on Autonomic Computing and Communication Systems, Autonomics 2007, 28-30 October 2007, Rome, Italy*, ser. ACM International Conference Proceeding Series, vol. 302. ACM, 2007, p. 15. [Online]. Available: https://doi.org/10.4108/ICST.AUTONOMICS2007.2113

[7] T. Vogel, J. Bruhn, and G. Wirtz, "Autonomous reconfiguration procedures for EJB-based enterprise applications," in *Proceedings of the Twentieth International Conference on Software Engineering & Knowledge Engineering (SEKE'2008), San Francisco, CA, USA, July 1-3, 2008*. Knowledge Systems Institute Graduate School, 2008, pp. 48–53.

[8] F. Boyer, O. Gruber, and G. Salaün, "Specifying and verifying the SYNERGY reconfiguration protocol with LOTOS NT and CADP," in *FM 2011: Formal Methods - 17th International Symposium on Formal Methods, Limerick, Ireland, June 20-24, 2011. Proceedings*, ser. Lecture Notes in Computer Science, vol. 6664. Springer, 2011, pp. 103–117. [Online]. Available: https://doi.org/10.1007/978-3-642-21437-0_10

[9] F. Boyer, O. Gruber, and D. Pous, "Robust reconfigurations of component assemblies," in *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 2013, pp. 13–22. [Online]. Available: https://doi.org/10.1109/ICSE.2013.6606547

[10] F. Durán and G. Salaün, "Robust and reliable reconfiguration of cloud applications," *J. Syst. Softw.*, vol. 122, pp. 524–537, 2016. [Online]. Available: https://doi.org/10.1016/j.jss.2015.09.020

[11] A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün, "Rigorous Design and Deployment of IoT Applications," in *Proc. of FormaliSE'19*. ACM, 2019.

[12] A. Krishna, M. L. Pallec, R. Mateescu, L. Noirie, and G. Salaün, "IoT Composer: Composition and Deployment of IoT Applications," in *Proceedings of the 41st International Conference on Software Engineering: Companion Proceedings, ICSE 2019, Montreal, Canada*. IEEE / ACM, 2019, pp. 19–22.

[13] J. Seeger, R. A. Deshmukh, V. Sarafov, and A. Bröring, "Dynamic IoT choreographies," *IEEE Pervasive Comput.*, vol. 18, no. 1, pp. 19–27, 2019. [Online]. Available: https://doi.org/10.1109/MPRV.2019.2907003

[14] 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 *Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice, ICSE (SEIP) 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 2019, pp. 131–140. [Online]. Available: https://doi.org/10.1109/ICSE-SEIP.2019.00022