Formal Verification of Device State Chart Models

Formal Verification of Device State Chart Models

Fulvio Corno
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
Email: fulvio.corno@polito.it

Muhammad Sanaullah
Dipartimento di Automatica e Informatica
Politecnico di Torino
Torino, Italy
Email: muhammad.sanaullah@polito.it

***Abstract*—Design and development of increasingly complex intelligent environments require rich design flows that include strong validation and verification methodologies. Formal verification techniques are often advocated, and they require formally described models of the smart home devices, their interconnections, and their controlling algorithms. Complete verification can only be achieved if all used models are verified, including individual device models. This paper proposes an approach to formally verify the correctness of device models described as UML State Charts, by checking their consistency with respect to the properties, declared in an Ontology, for the categories to which each device belongs. The paper describes the verification methodology and presents some first verification results.**

***Index Terms*—Formal Verification, Intelligent Environment, Smart Home, State Charts, Model Checking**

I. Introduction

Intelligent Environments (IEs) consist of different integrated heterogeneous devices, ranging from simple sensors to multi-feature devices [1]. The objective of IE designers is to create an environment which can take decisions and, as a result, the devices can perform their ordinary activities in an intelligent manner, by facilitating the environment users with ease, comfort, security and safety in their life [2]. The increasing complexity of IEs calls for the adoption of structured and formal design and verification techniques [3], using model-based approaches and formal validation and verification tools. In particular, all devices, their interconnections, and their governing algorithms should be modeled, to allow full system verification. This paper focuses on the correctness of the formal models used to describe the individual smart devices, by ensuring the consistency between declarative and operational representation. The paper complements previous work from the same authors [4] where interconnections and governing algorithms were considered and verified.

In modeling smart devices, both device interface and behavior information are considered. *Interface* information describes the device as a black box, listing functionalities (a device can perform), commands (that can be sent to the device), notifications (the device can send) and states (in which a device can be at a specific time). The detailed information about the internal workings of the device, such as the action performed when an command is received in a particular state, falls into the *behavior* category. Interface and behavior descriptions can be expressed as a joint model (like in [5]), or with separate models (like in our previous work [6], [7]).

DogOnt [6] is an Ontology based solution for interface modeling and reasoning about such devices, whereas the behavior of each device is modeled in a Device State Chart (DSC) [7]. According to Harel [8], state charts are best suited for representing the complex behavior of communicating devices. In state charts, the internal behavior of devices is represented with states, functionalities, transitions (composed of source state, destination state, triggers, guards and actions) and variables.

In our previous work [4], we proposed a design time methodology to formally verify the correct behavior of IEs, by checking the intended system properties. The methodology adopted a model checking approach for verifying the temporal properties over a model of the entire IE, which consists of state charts of devices and control algorithms. Control algorithms are the software components which intelligently control the overall operations of IE.

As with any design artifact, the DogOnt ontology and the Device State Charts are prone to errors and inconsistencies. In our previous work [4], we verified the correctness of IEs on the basis of these DSCs, but the correctness of the latter was not checked at that time.

This paper addresses and solves the problem by proposing a formal methodology, based on model checking, for the automatic identification of inconsistencies between device information available in DogOnt and the behavior of the corresponding DSC. These inconsistencies must be manually fixed and then the verification process is repeated until consistency is reached.

The adopted technologies are presented in section II, and in particular the previous work, on which the proposed approach is based, is detailed in section II-F. The proposed approach for DSC verification is presented in section III, and a case study with the implementation details is presented in section IV. Results and remarks about the case study are given in section V and conclusions are finally drawn in section VI.

II. Background

Our approach uses state machines to represent devices behavior, and is based on a well established smart home framework that adopts semantic web technologies –in the form of an Ontology– known as DogOnt. It uses the UMC model checker [9] for identifying the inconsistencies between DogOnt and DSC. For expressing properties, it uses UCTL temporal properties. The details of the mentioned concepts are

given in the following subsections with an example of device modeling in DogOnt.

A. DogOnt in a nutshell

In IEs, mostly, hardware (devices) and software (control algorithms) components interact with each other in an intelligent manner. The modeling of these devices (hardware components) can be performed by adopting different techniques. As these devices are of various heterogeneous types, researchers adopted Ontologies for modeling of such systems [10]. Ontologies are one of the semantic web artifacts; they are based on graph like structure, for representing different concepts, their relationships and their associations, and give a suitable reasoning power on these. DogOnt [6] is an Ontology based solution for the modeling of Smart Homes (IEs).

DogOnt identifies the following basic dimensions for the modeling of IEs (Figure 1):

1) "Building Environment": information about the environment where the IE is implemented, like building, flat, garage, garden, room (bathroom, bedroom, dining-room, kitchen, living-room, etc);
2) "Building Things": information about the physical objects used in the IE, among which some are electrically controllable. The electrically controlled devices (like coffee maker, boiler, cooker, fan, lamp, actuator, sensors and various others), used in IE, are categorized under the Controllable category. Physical objects (like table, sofa, wall, ceiling and others), which can not be electrically controlled, are categorized under the Uncontrollable category.
3) "Functionality": Controllable devices used in IEs have different functionalities. DogOnt further classifies them as Control Functionality, Notification Functionality or Query Functionality. Control functionalities are the actions that devices can perform (like a Lamp has "on" and "off" functionalities). Most controllable devices have the ability to send a notification back, as an acknowledgment of the completion of the assigned task. These notification capabilities are modeled under the Notification functionalities classification. For performing the task intelligently, usually, it is required to have a mechanism through which the status of devices can be queried at any time, which are modeled under the Query functionalities classification;
4) "Commands": Controllable devices in IEs perform their Control functionalities by receiving some particular commands, whose modeling and classification is performed under this category;
5) "Notification": Controllable devices send different types of notifications, e.g., when their state changes;
6) "State" and "State Values": Controllable devices possess an internal state, which is modeled a set of orthogonal state spaces (called "States"), with different "State Values" each. A complete description of the state of a device is therefore a valid State Value for each of the States defined for that device. State Values can be discrete (e.g., in Lamp "onState" and "offState") or continuous within a specific range of values (e.g, a DimmerLamp device has a "Light-Intensity-State" whose value ranges from 0% to 100%).
7) "Domotic Network Component": Controllable devices adopt widely different network protocols. This modeling dimension describes the protocol characteristic and network addressing scheme.

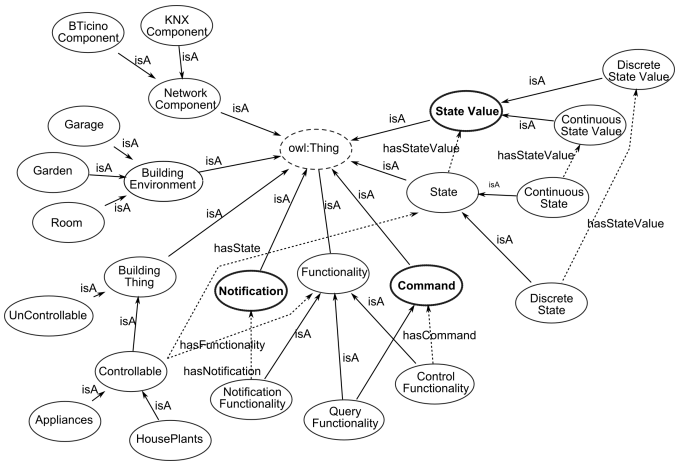

Figure 1. DogOnt

B. Device Modeling in DogOnt

According to the DogOnt classification dimensions, each device is modeled by creating instances for all relevant DogOnt classes, and according to the ontology constraints. The device modeling process is briefly explained by illustrating the model of a "Dimmer Lamp" device (Figure 2), which is a subclass of "Lamp" and "Controllable", and has all inherited features of these super classes.

A dimmer lamp has all the functionalities which a "Lamp" can hold, such as, it can be (switched) "on" and "off", and can be placed at a certain location in the IE. Furthermore, "Dimmer Lamp" has an extra "Light-Regulation-Functionality" by which the "Light-Intensity" of the lamp can be managed. The value of "Light-Intensity" ranges from 0 to 100. With the "Light-Regulation-Functionality" it may be increased or decreased with a step 10 through "stepUp" or "stepDown" commands respectively, or it may also be directly set to a specific value with "set(value)" command. As "Dimmer Lamp" is a type of Controllable device, two more functionalities are inherited from the class of "Controllable", these are "Query-Functionality" and "State-Change-Notification-Functionality".

More than 143 device classes are currently modeled in DogOnt.

C. State Charts

In 1987, Harel [8] extended the semantics of state-transition diagram for specifying the complex behavior of reactive system and named it State Charts, whose variant, afterward, became a standard in Unified Modeling Language (UML) 2.0.

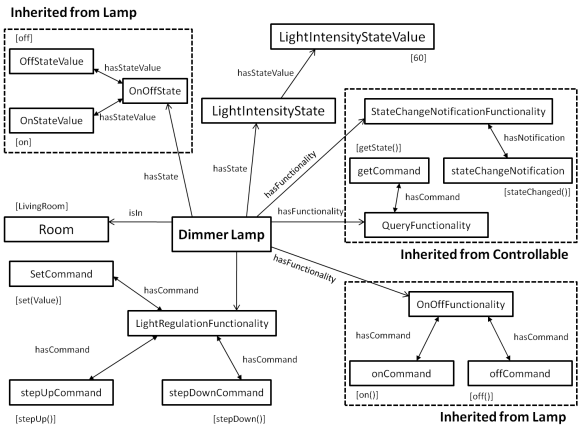

Figure 2. Dimmer Lamp in DogOnt

In reactive systems, internal and external communication in-and-between different devices is performed through message-exchange.

DogOnt only contains the interfaces information (functionalities, commands, notifications, states and others) of devices, and this is sufficient to interact with the device. But the behavior of devices must be modeled with an operational representation, such as State Charts. Device behavior is encoded as a set of transitions among internal states. Each transition between a source state S_S and a destination state S_D is represented as $S_S \rightarrow S_D(T[G]/A)$, where T (trigger) is an event (command, notification) which a device can receive when in S_S, and provided that the guards G (boolean conditions) are satisfied, the action A (command, notification, expression evaluation or other) is performed, by also switching to S_D.

D. Devices State Charts

For representing the behavior of IE devices, we use Device State Charts (DSCs), which are encoded in SCXML (State Chart XML) [11]. The "Dimmer Lamp" state chart, in SCXML syntax, is given in Figure 3 and the corresponding Harel state chart is presented in Figure 4. In SCXML, DSCs work as Templates for specified devices; the "id" attribute is used for containing the name of device instances.

The presented DSC shows the behavior of "Dimmer Lamp": it contains "off", "on" and "LightIntensityState" states, where "off" is the initial state, and "LightIntensityState" is a sub-state of "on". With "on()" or "set(value)" commands, "Dimmer Lamp" switches its state from "off" to "on". When it receives the "on()" command, it is switched on and moves to "LightIntensityState", and when it receives 'set(value)" command, the lamp is switched on, the intensity of the light is set with the parametric value of this command and the state moves to "LightIntensityState", where the intensity of light can be controlled (details are given in II-B). When the "off()" command is received, the state is switched from "LightIntensityState" to "off". For storing the "Light-Intensity" value in DSC, a variable named "lightIntensity", with 50 as initial value, is used, and "lightStep" is used with the value of 10 for increasing or decreasing the light intensity.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE scxml SYSTEM "template.dtd">
<!-- @device=DimmerLamp -->
<scxml xmlns="http://www.w3.org/2005/07/scxml"
version="1.0">

<state id="&id;dimmerLamp">
<initial> <transition target="&id;off"/> </initial>

<datamodel>
  <data name="&id;lightIntensity" expr="50.0"/>
  <data name="&id;lightStep" expr="10.0" />
</datamodel>

<state id="&id;off">
  <transition event="&id;on" target="&id;on" />

  <transition event="&id;set"
    target="&id;lightIntensityState">
    <assign name="&id;lightIntensity"
      expr="_&id;set.value" />
  </transition>
</state>

<state id="&id;on">
<initial>
  <transition target="&id;lightIntensityState" />
</initial>

<transition event="&id;off" target="&id;off" />

<state id="&id;lightIntensityState">
  <transition event="&id;stepUp"
    target="&id;lightIntensityState">
    <assign name="&id;lightIntensity"
      expr="if(&id;lightIntensity lt 100.0)
      {&id;lightIntensity + &id;lightStep;}
      else{100.0;}" />
  </transition>

  <transition event="&id;stepDown"
    target="&id;lightIntensityState">
    <assign name="&id;lightIntensity"
      expr="if(&id;lightIntensity gt 0.0)
      {&id;lightIntensity - &id;lightStep;}
      else{0.0;}" />
  </transition>

  <transition event="&id;set"
    target="&id;lightIntensityState">
    <assign name="&id;lightIntensity"
      expr="_&id;set.value" />
  </transition>
</state> <!--lightIntensityState-->
</state> <!--on-->
</state> <!--dimmerLamp-->
</scxml>
```

Figure 3. The State Chart of Dimmer Lamp in SCXML format

E. UMC Model Checker

Model checking is a technique used for automatically verifying the behavior of the system according to its specifications. It is capable of exhaustively considering all the states of the model for checking the correctness of specification. UMC [9] is an "on-the-fly" model checker tool, designed for the formal verification of the dynamic behavior of UML state charts, by providing a user friendly environment for expressing the system and the properties. UMC is fast because it is based on a linear time complexity model checking algorithm for the exact verification of the system.

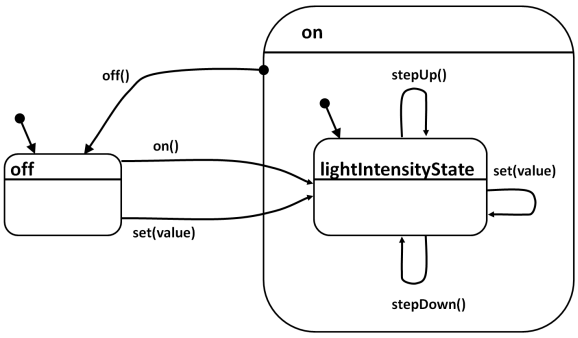

Figure 4. Harel State Chart of a Dimmer Lamp

The structure of the model verified by UMC consists of classes, instances and abstraction rules. Classes are used to represent the state machines in textual format. They have states, operators (used for synchronous communication of messages) or signals (used for asynchronous communication of messages), local variables and transitions. Further, transitions are associated with states (source and destination), triggers, guards and actions. Instances are the objects of the classes. Abstractions are the mechanism for representing the subset of relevant information, which users want to observe, from the large amount of states and actions (operators and signals) in the complex graph of the modeled system.

An on-line version of the UMC model checker is also available[1].

The requirements and specifications of the systems must be written in some formal way, so that, they can be verified on the model of the system. Temporal logic is one of these formal ways, which is best suited for the verification of reactive systems, as it is a system of rules for reasoning with different propositional quantifiers [12].

For the verification of complex systems, UMC supports the UCTL branching time temporal logic [13], which is a UML-oriented branching-time temporal logic, with the combined power of ACTL (Action Based Branching Time Logic) [14] and CTL (State Based Branching time logic) [15]. UCTL uses the box [ ] ("necessarily") and diamond <> ("possible") operators from Hennessy-Milner Logic [16] and temporal operators (Until, Next, Future, Globally, All, Exists) from CTL/ACTL.

Due to the rich set of state propositions and action expressions, UCTL is best fitted for the verification of state machines. With the help of UCTL, we can verify different properties like liveness (something good will eventually happen), safety (nothing bad can happen) and properties with or without the fairness restrictions.

F. Verification methodology for IEs

In our previous work [4], we suggested a design time methodology for the verification of IEs, by adopting a Model Checking approach, based on DogOnt, DSCs and Temporal properties.

[1]http://fmt.isti.cnr.it/umc/

Temporal properties are designed in order to verify the specifications of IE by checking the existence, absence and sequence of commands, notifications or states in IE. The state charts of IE model (devices and control algorithms) are given to the Model Checker and Temporal properties are verified on them. In case any specification is found incorrect, the required modification is performed and the verification process is repeated until all the specifications satisfy. The approach is presented in Figure 5, and shows that the model under verification consists of a collection of State Charts obtained by combining all the individual DSCs, according to IE configuration and topology.

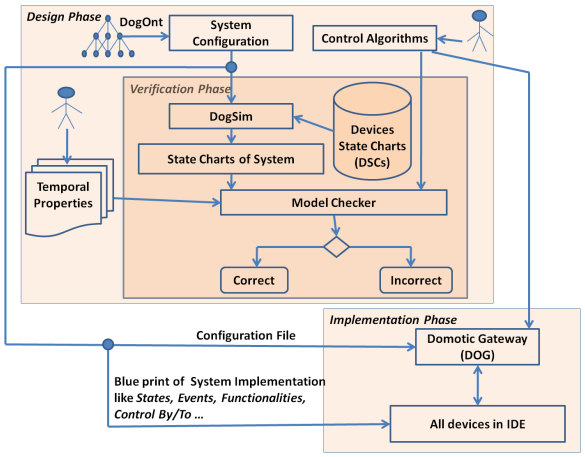

Figure 5. Design Time Methodology

Although the above methodology verifies the correctness of IE, a missing point, which was not considered in [4], is the correctness of these DSC. These should model the actual device behavior and at the same time be consistent with the information available in DogOnt: there is a likelihood of inconsistencies and discrepancies between DSC and DogOnt. In the following proposed approach, we verify these DSCs and ensure their consistency with DogOnt.

III. Proposed Approach

The goal of this paper, as mentioned earlier, is to prove the consistency between DogOnt (interface information) and the corresponding DSC (behavioral information) for all devices classes, by identifying and fixing discrepancies. In DSCs, each state can accept some messages, depending on the outgoing transitions; anything other than them, will be rejected on that state and no action will be performed. The action is in the form of some internal activities or commands/notifications to other DSCs. The following type of discrepancies may occur:

1) the DSC may lack or have some extra notifications with respect to the ones modeled in DogOnt;
2) the DSC may not receive a command/notification which is modeled in DogOnt for fulfilling the required functionalities of the device;
3) the DSC may not have a state which is modeled in DogOnt;

4) the DSC may follow some behavior which may not conform with the specifications;
5) the DSC may be designed in such a way that a deadlock may occur.

The proposed approach, whose general architecture is depicted in Figure 6, consists of three components, namely "Temporal Properties Generator" (TPG), "Environment Designer (ED)" and "Model Builder (MB)". TPG and ED are fully automatic, whereas MB is semi-automatic.

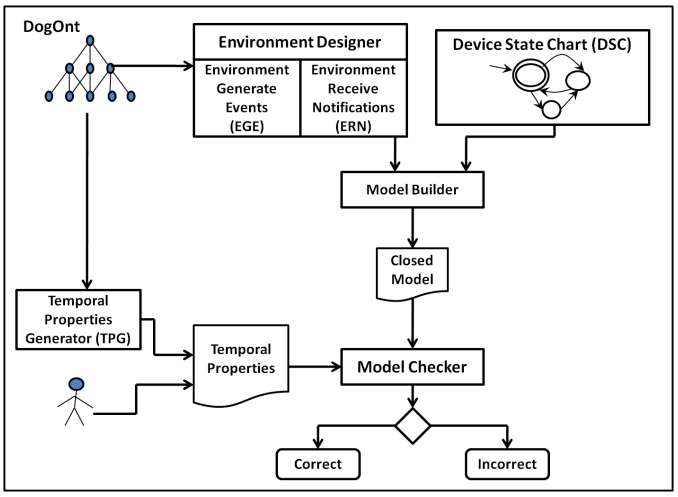

Figure 6. Library Files Verification Methodology

Ontologies (such as DogOnt) and State Charts (such as DSCs) are not directly comparable formalisms. In our verification approach, we extract some logic and temporal properties from DogOnt, and we check them against the device DSC. Such properties are derived from the semantic description of the device. These temporal properties are of the following types:

1) the acceptance of all the described commands;
2) the generation of all the described notifications;
3) the reachability of all the described states.

If needed, for checking, e.g., the correct behavior, extra information or deadlock in DSC, our approach also supports manual design of further temporal properties.

Verification of a device state chart (DSC) model requires to embed it into a suitable environment, that may generate and receive the appropriate events. For each DSC, we automatically generate an "Environment" in which it is embedded, so that the Environment plus the DSC make a closed system. The environment expects the DSC to implement the interface (commands, notifications, states) prescribed in DogOnt, and is designed to be as hostile as possible (generating commands with arbitrary orders and speeds), and as general as possible (allowing all possible device behaviors and execution paths): in this way, the verification results will be valid for any possible "real" environment in which the device may be embedded.

The Environment can send all the commands to DSC and can receive the notifications from it, where the information of commands and notifications is obtained from DogOnt. Its functionality is represented in Figure 7 and is composed of an "Environment Generate Commands" (EGC) and an "Environment Receive Notification" (ERN) parallel modules. EGC generates all the commands obtained from DogOnt, where as ERN receives all the notifications modeled in DogOnt against this device.

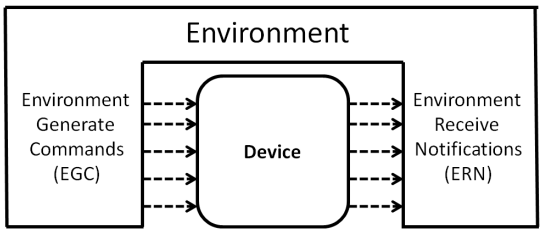

Figure 7. Environment behavior in Verification Process

The Environment is implemented by an "Environment Designer" component, which queries DogOnt and automatically generates the state charts of EGC and ERN in a format acceptable by the model checker. After generating these, it sends them to "Model Builder", which performs the following activities;

a. semi-automatically translates DSC from SCXML to the language supported by the model checker. The current "SCXML to UMC" converter tool only deals with those SCXML tags used in the modeling of device behavior;
b. automatically combines the Environment state charts (EGC and ERN) with the state chart of the device;
c. automatically adds abstraction rules and generates instances of these state machines, so that UMC code is ready for further processing;
d. automatically saves all these in a file known as "Closed Model".

The Temporal Property Generator (TPG) module obtains the information about device states, commands and notification from DogOnt, and it automatically generates three types of properties in the form of UCTL logic:

1) the first property group checks that all the commands declared for the device are actually sent by the EGC component, and that all the notifications are eventually sent by the device;
2) the second group checks that such messages are actually received: commands are correctly received by the device DSC and notifications are correctly received by ERN;
3) the third type of properties verifies the reachability of all states that are explicitly mentioned in DogOnt.

The Closed Model and these Temporal Properties are sent to the Model checker, which is responsible for the verification of these temporal properties on the given model. This verification process is repeated until all found inconsistencies are resolved. Advanced properties may be manually added to verify the correct behavior of the modeled device.

Device	# External Commands	# DogOnt Commands	# DogOnt Notifications	# DogOnt States	# Explored States (max)	# Automatically designed TPs	# Satisfied Properties
Button	1	0	3	2	16	8	8
Dimmer Lamp	0	5	1	3	417	15	13
Door Actuator	4	2	1	4	65	10	8
Infrared Sensor	2	0	3	2	16	8	8
Mains Power Outlet	0	6	1	2	30	16	12
On Off Switch	1	0	3	2	12	8	8
Shutter Actuator	2	3	1	5	50	13	11
Simple Lamp	0	2	1	2	14	8	8
Smoke Sensor	2	0	3	2	16	8	8
Toggle Relay	0	1	3	2	12	10	10
Window Actuator	4	2	1	4	65	10	8

Table I
LIST OF VERIFIED DEVICE STATE CHARTS (DSCS) THROUGH THE PROPOSED APPROACH

IV. CASE STUDY

The proposed approach (section III) relies on tools (shown in Figure 6) implemented in Java. For querying from the Ontology, "Temporal Properties Generator" (TPG) and "Environment Designer (ED)" use the Jena libraries [17], while the "Model Builder (MB)" reads DSCs through an XML DOM parser.

Eleven Device State Charts (DSCs), listed in Table I, were verified by this approach. The table also gives some information about the DogOnt modeling of each device, the number of automatically generated properties, and the number of properties found true by the model checker. The last columns show that some devices do not comply with the specifications, and the DSC (or possibly the DogOnt model) should be modified. Computation times were reasonable, and they amount to just some seconds per each verified property.

The complete case study of a Dimmer Lamp is presented here, for illustrating the details. DogOnt currently models the Dimmer Lamp class through the set of attributes (direct and inherited) listed in Table II. According to the modeled information, it has two types of states: "OnOffState" and "lightIntensityState". "OnOffState" is a discrete value state with "on" and "off" as state values, whereas "lightIntensityState" is a continuous value state, in the range of 0 to 100 and is used for managing the intensity of light. A dimmer lamp has two types of command functionalities: "OnOffFunctionality" and "lightRegulationFunctionality". "OnOffFunctionality" is used for switching the Dimmer Lamp "on" and "off", and "lightRegulationFunctionality" is used for increasing or decreasing or even setting the intensity of light in Dimmer Lamp to a specific value (obtained through "Level-Control-Functionality", which is in the range of 0 and 100). It has one notification functionality, "StateChangeNotificationFunctionality"; which sends a notification when the state is changed.

With the above information, the properties about the reachability of states and the existences of the path (commands) are generated automatically by the TPG block. For each command or notification we generate two properties, one checking that the corresponding event is sent, and the other checking that it is accepted by the destination state chart. State properties are also automatically designed for verifying the existence and reachability of all the states which are modeled in DogOnt. The automatically designed properties for the Dimmer Lamp are reported in Figure 8 in UCTL syntax.

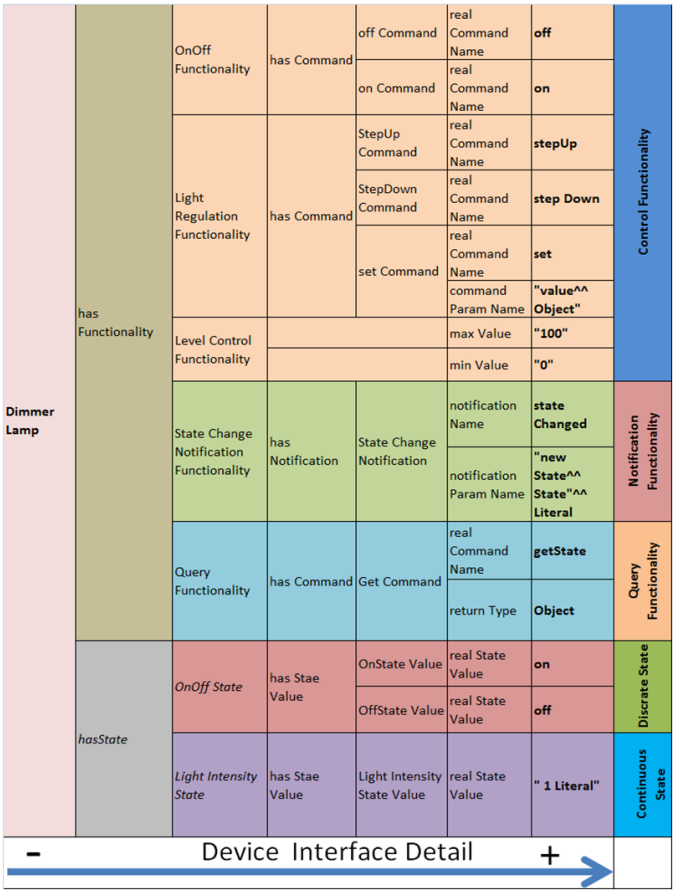

Table II
DIMMER LAMP DETAIL AVAILABLE IN DOGONT

With the information presented in Table II, EGC and ERN state charts are designed automatically and sends them to the Model builder. Model Builder semi-automatically obtains the translated state chart of the DSC by using our customized

```
--Action Properties
  --the acceptance of all the commands in DSC
    EF {sending(stepDown)} true
    EF {sending(stepUp)} true
    EF {sending(set)} true
    EF {sending(off)} true
    EF {sending(on)} true
  --
    EF {accepting (stepDown)} true
    EF {accepting (stepUp)} true
    EF {accepting (set)} true
    EF {accepting (off)} true
    EF {accepting (on)} true

  --the generation of all the notifications in DSC
    EF {sending(stateChanged)} true
    EF {accepting(stateChanged)} true

--State Properties
  --the reachability of all the states in DSC
    EF (offState)
    EF (onState)
    EF (LightIntensityState)
```

Figure 8. Automatically designed Temporal Properties for Dimmer Lamp

SCXML to UMC converter. It then adds required abstraction statements for renaming states and commands. Some abstractions are added for checking the bounds of light intensity values. Currently this step is manual and we are automating it. Finally the model builder creates the instances of these state machines, so that the model can be ready for verification processing. The full Dimmer Lamp model generated by Model Builder, including EDC and ERN, is shown in Figure 10.

V. Results and remarks

The Model Checker, after checking these properties (shown in Figure 8) on the designed model (shown in Figure 10), found that the properties related to "StateChangeNotification-Functionality" were false. When the model was analyzed, it was observed that the "StateChangeNotificationFunctionality" was not modeled in the DSC. Necessary actions were taken and the DSC of Dimmer Lamp was modified. After the modification, the process was repeated and all properties were found true.

```
E [true {not accepting(off)} U
       {accepting(on) or accepting(set)} true]
E [true {not (accepting(stepDown)or accepting(stepUp))} U
       {accepting(on) or accepting(set)} true]
EF (not underFlow)
EF (not overFlow)
EF (inRange)
```

Figure 9. Manually designed Temporal Properties for Dimmer Lamp

Furthermore, some advanced properties for verifying the correct behavior of Dimmer Lamp were manually added, shown in Figure 9: the first two properties check that "off", "stepUp" and "stepDown" commands are not be acceptable unless the Dimmer Lamp is in the "on" state. The last three properties check the boundary values of the light intensity state value, thanks to the definition in the closed model of the abstactions for "underFlow," "overFlow" and "inRange" conditions. The model checker found the last three properties to be false: the analysis of the DSC showed that the condition `lightIntensity < 100` should be replaced by `lightIntensity + lightStep <= 100`, and similarly for the decrement. After this final correction, the verification process was repeated all properties were found true. Thus now, the Dimmer Lamp model (DSC) is consistent with DogOnt and behaves well according to the requirements.

VI. Conclusion and Future Work

IEs can be modeled by adopting different approaches; DogOnt is one ontological solution among them. In DogOnt, interface information of devices are modeled,and the behavior of the devices are modeled in Device State Charts (DSC). There is a fair chance of missing and/or providing extra information and inaccuracies, as DogOnt and DSCs are heavily based on human observation and manual input, influenced by a huge variety of devices, extensible and personalize-able, creating inconsistencies between DogOnt and DSC. The proposed methodology is about finding and fixing these discrepancies and making DSCs consistent with DogOnt.

Our future work is about finding the sequences of transitions, which are required to be followed, for letting the IEs reach a desired and set configuration. This configuration may be simple (based on the behavior of one device) or complex (based on the complex behavior of different devices). By finding the sequence of transitions, one can know the way that (s)he can follow to achieve the desired configuration of environment.

Acknowledgements

This work is partially supported by the Higher Education Commission (HEC), Pakistan under UESTP-Italy/UET project. The authors thank Franco Mazzanti for his guidance and support.

References

[1] H. Ristau, "Publish/process/subscribe: Message based communication for smart environments," in *IET 4th International Conference on Intelligent Environments*, 2008, pp. 1–7.

[2] K. Ducatel, M. Bogdanowicz, F. Scapolo, J. Leijten, and J.-C. Burgelman, "Scenarios for Ambient Intelligence in 2010," ISTAG: IST Advisory Group, Tech. Rep., February 2001.

[3] A. Coronato and G. De Pietro, "Formal design of ambient intelligence applications," *Computer*, vol. 43, no. 12, pp. 60 –68, Dec. 2010.

[4] F. Corno and M. Sanaullah, "Design time Methodology for the Formal Verification of Intelligent Domotic Environments," in *Ambient Intelligence - Software and Applications*, ser. Advances in Intelligent and Soft Computing, P. Novais, D. Preuveneers, and J. Corchado, Eds. Springer Berlin / Heidelberg, 2011, vol. 92, pp. 9–16.

[5] D. Roman, M. Kifer, and D. Fensel, "Wsmo choreography: from abstract state machines to concurrent transaction logic," in *Proceedings of the 5th European semantic web conference on The semantic web: research and applications*, ser. ESWC'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 659–673.

[6] D. Bonino and F. Corno, "DogOnt - Ontology Modeling for Intelligent Domotic Environments," in *International Semantic Web Conference*, ser. LNCS, A. Sheth, S. Staab, M. Dean, M. Paolucci, D. Maynard, T. Finin, and K. Thirunarayan, Eds., no. 5318. Springer-Verlag, October 2008, pp. 790–803.

```
Class State is
end State;

Class EGC is
  Vars: RandomValue:int=35
  State top = E
  Transitions:
    E -> E {-/DimmerLampInstance.set(RandomValue)}
    E -> E {-/DimmerLampInstance.stepDown()}
    E -> E {-/DimmerLampInstance.stepUp()}
    E -> E {-/DimmerLampInstance.off()}
    E -> E {-/DimmerLampInstance.on()}
end EGC;

Class ERN is
  Operations: stateChanged(newState:State)
  State top = N
  Transitions:
    N -> N {stateChanged(newState)/}
end ERN;

Class DimmerLamp is
  Operations: on(), off(),set(value:int),
    stepUp(), stepDown()
  Vars: lightIntensity:int=50, lightStep:int=10
  State top = off, on
  State on = lightIntensityState

  Transitions:
    off-> on{on()/ }
    off-> lightIntensityState{set(value)/
          lightIntensity:=value}

    on -> lightIntensityState{-/}
    on-> off{off()/ }

    lightIntensityState ->  lightIntensityState{stepUp() /
      if (lightIntensity < 100 )then
      {lightIntensity :=  lightIntensity + lightStep}
         else {lightIntensity :=  100}; }

    lightIntensityState ->  lightIntensityState{stepDown() /
        if (lightIntensity > 0)then
         {lightIntensity :=  lightIntensity - lightStep}
         else {lightIntensity :=  0}; }

    lightIntensityState ->  lightIntensityState{set(value)/
                            lightIntensity:=value}
end DimmerLamp

Objects:
ec: EGC
en: ERN
DimmerLampInstance: DimmerLamp

Abstractions{
 Action $1($*) -> $1($*)
 Action $1 -> sending($1)
 Action accept($1) -> accepting($1)
 Action lostevent($1) -> discarding($1)

 State inState(DimmerLampInstance.lightIntensityState)
   -> LightIntensityState
 State inState(DimmerLampInstance.off) -> offState
 State inState(DimmerLampInstance.on) -> onState

 State DimmerLampInstance.lightIntensity < 0 -> underFlow
 State DimmerLampInstance.lightIntensity > 100 -> overFlow
 State DimmerLampInstance.lightIntensity >= 0 and
       DimmerLampInstance.lightIntensity <= 100 -> inRange
}
```

Figure 10. Complete Model File of the Dimmer Lamp

[7] ——, "DogSim: A State Chart Simulator for Domotic Environments," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on*, 29 2010-april 2 2010, pp. 208 –213.

[8] D. Harel, "Statecharts: a visual formalism for complex systems," *Science of Computer Programming*, vol. 8, no. 3, pp. 231 – 274, 1987.

[9] F. Mazzanti, "Designing uml models with umc," Technical Report 2009-TR-43, ISTI-CNR-Pisa, Italy, Tech. Rep., 2009.

[10] D. Fensel, *Ontologies: a silver bullet for knowledge management and electronic commerce*. New York, NY, USA: Springer-Verlag, 2001.

[11] J. Barnett et al., "State chart XML (SCXML): State Machine Notation for Control Abstraction," W3C, Tech. Rep., May 2010.

[12] Z. Manna and A. Pnueli, *The temporal logic of reactive and concurrent systems*. New York, NY, USA: Springer-Verlag New York, Inc., 1992.

[13] F. Mazzanti, *UMC 3.3 User Guide, ISTI Technical Report 2006-TR-33*, ISTI-CNR Pisa-Italy, September 2006.

[14] R. De Nicola and F. Vaandrager, "Action Versus State based Logics for Transition Systems," *Semantics of Systems of Concurrent Processes, Lecture Notes in Computer Science*, vol. 469, pp. 407–419, 1990.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications," *ACM Transactions on Programming Languages and Systems*, vol. 8:2, pp. 244–263, April 1986.

[16] M. Hennessy and R. Milner, "On observing nondeterminism and concurrency," in *Automata, Languages and Programming*, ser. Lecture Notes in Computer Science, J. de Bakker and J. van Leeuwen, Eds. Springer Berlin / Heidelberg, 1980, vol. 85, pp. 299–309.

[17] B. McBride, "Jena: A semantic web toolkit," *IEEE Internet Computing*, vol. 6, pp. 55–59, November 2002.