A Tool to Edit and Verify IoT System Architecture Model

Shinpei Ogata*, Hiroyuki Nakagawa[†], Yoshitaka Aoki[‡], Kazuki Kobayashi*, Yuko Fukushima[‡] * Shinshu University, Nagano, Japan Email: ogata@cs.shinshu-u.ac.jp, kby@shinshu-u.ac.jp [†] Osaka University, Osaka, Japan Email: nakagawa@ist.osaka-u.ac.jp [‡] Nihon Unisys, Ltd., Tokyo, Japan Email: {yoshitaka.aoki, yuko.fukushima}@unisys.co.jp

Abstract—The spread of IoT (Internet of Things) categorized into a type of cyber-physical system makes future systems larger and more complex than ever. Various components such as cloud services, edges, devices and energy suppliers play important roles when constructing an IoT system. Moreover, components in such systems have various relationships to other components. Hence, a simple and extendable specification notation for grasping such relationships is sought for architectural design of IoT systems. When we design such IoT Systems, effective verification also should be provided so that we can identify critical system faults. We present a tool to edit and verify IoT system architecture models. The tool extends a UML editor to enable easy editing of architectural models, and verifies them using the logic programming language Prolog.

Keywords-IoT, Architectural Model, Logic Programming, Verification, Twin Peaks Model.

I. INTRODUCTION

IoT systems are often developed as large-scale systems because they have to deal with not only cyberspaces but also physical spaces. Cyberspaces contain cloud services storing and utilizing big data, whereas physical spaces contain devices interacting with things outside of the system and equipment supplying energy to other pieces of physical equipment. Therefore, system faults occur frequently in such IoT systems than ones in traditional software centric systems. Developers should comprehend target IoT systems that contain various components and their mixed and complicated relationships.

The twin peaks model [1], [2] describes a method to develop software requirements and architectures concurrently but separately using incremental development. This model is still efficient when we construct an IoT system. Goal models [3], [4] and use case models [5] are representative models for requirements analysis. The goal models, such as i* [6], can represent actor relationships; the use case models, on the other hand, mainly represent use-case-centric relationships instead of actor relationships. We can analyze intentions and features using goal models and use case models; however, when we design a system architecture, it is difficult to comprehend various relationships between cyber objects and physical objects. In this paper, architectural objects are defined as a super concept of components and actors..

We propose TORTE, which is a method of modeling IoT system architectures. In TORTE, architectural objects, their relationships and their properties are captured in order to model an IoT system architecture. TORTE maps each relationship into a corresponding single layer, and widely captures such intra- and inter-relationships between objects. Each type of relationship has a respective description space called a layer so that the complexity can be mitigated even if many relationships which may be different from each other are defined. A layer is also placed in an architectural view at an early stage of the development conforming to the twin peaks model. We present a novel tool which provides two main features for supporting TORTE:

- An editor for creating a layered model of an IoT system architecture using six types of architectural objects and relationships.
- A verifier for checking the consistency of the model using logic programming.

In this paper, we experimentally model an existing farm monitoring system, which is an IoT system, using the tool. The monitoring system has various physical objects such as a camera, solar panel, and battery, and cyber objects such as an original monitoring service and some SNSs (Social Network Services). We demonstrate our tool by applying it to the model of the monitoring system.

II. DESIGN MODEL FOR IOT SYSTEM ARCHITECTURES

Our architectural model consists of a set of six structural diagrams in a variant UML class diagram notation. The model also provides six types of architectural objects (hereafter simply called objects) shown in Table I and directed relationships (hereafter simply called relationships) shown in Table II.

Among these objects, energy suppliers do not often appear in general software design, but play an important role when we consider the sustainability and dependability of IoT systems. In particular, when we design an IoT system under unreliable energy suppliers, the amount of energy consumption and the lack of energy should be considered. Therefore, we have captured energy suppliers as energy objects. Figure 1 shows a model example and the tool. The tool is introduced in detail in Section IV. Figure 1(a) shows an example of layers regarding transmit-data and transmit-energy in the farm monitoring system. A rectangle, i.e. class, represents an object and an arrow, i.e. association, represents a relationship. A string between << and >>, i.e. stereotype, represents the type of object or relationship.

An object is defined as an actor or a component of a system under development (SUD). The UML 2.5 specification [5]

TABLE I The types of architectural objects

Туре	Description	
user	Users using or maintaining other objects.	
service	Services provided for other objects and deployed in cyberspace.	
edge	Edges constructing networks and deployed in physical space.	
device	Devices controlling or monitoring other objects and deployed in physical space.	
energy	Energy suppliers transmitting energy for other objects and deployed in physical space.	
environment	Natural environment stimulating other objects or monitored them.	

TABLE II THE TYPES OF ARCHITECTURAL RELATIONSHIPS

Туре	Description	
use	An object uses a beneficial service of another object.	
request	An object requires data from another object.	
control	An object changes or maintains behaviors of another actor.	
monitor	An object monitors another object without requiring a response.	
transmit-data	An object transmits data to another object.	
transmit-energy	An object transmits energy to another object.	

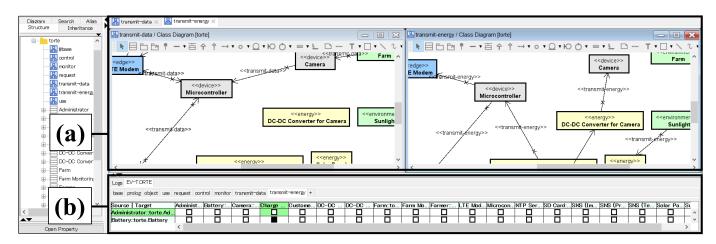


Fig. 1. A model example and the tool

defines an actor as follows: "*an actor specifies a role played by a user or any other system that interacts with the subject.*" That is, actors interact with a SUD from the outside and their capability cannot be changed by the developers of the SUD.

Meanwhile, components behave as a part of a SUD and their capability can be changed by the developers of the SUD. When use cases and their relationships are given to models, the models have the aspect of a use case diagram [5]. Hence, the models have an architectural view in the twin peaks model [2], [1] when use case models are assumed to be a requirements view.

According to existing research [7], [8], [9] on IoT system development, relationships have been variously categorized but are different from each other. Thus, an architectural design model should have the flexibility to easily accept new viewpoints, i.e. relationship types. TORTE can easily accept new viewpoints by adding corresponding layers without affecting existing layers.

III. MODEL VERIFICATION

Fig 2 shows the overview of the model verification process. This research assumes that architectural elements, i.e. objects and relationships, have common behaviors for each type of element. For instance, the behaviors include interaction protocols when controlling, monitoring or transmitting something. The combinations of such behaviors across different objects should be verified at an early stage of the design phase. Hence, our method employs the logic programming language Prolog for the verification.

According to the ISO/IEC standard of Prolog [10], A program in Prolog consists of many clauses. Each clause is categorized into either a *fact* or a *rule*. *Fact* clauses can be completely generated from model elements in TORTE. Table III shows the correspondences between model elements and the resulting *fact* clauses. Bracketed strings, e.g.

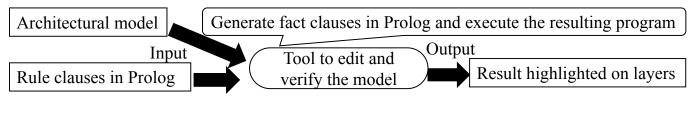


Fig. 2. Overview of the model verification process

 TABLE III

 CORRESPONDENCES BETWEEN MODEL ELEMENTS AND fact CLAUSES

Model elements	Fact clauses
Each object	object([name], [type]).
Each relationship	r([type], [source], [target]).
Each object property	[property's name]([object's type], [object's name]).

[name], represent variables which are replaced with model elements or their properties in the generation. For instance, a camera object categorized as device is transformed into object (camera, device).

Meanwhile, rule clauses pre-defined by TORTE represent common behaviors using model elements and their properties. Listing 1 shows the *rule* clauses excerpted. For instance, the clause *notEnergySupplied(X, Ls)* aims to discover whether there is a path in which an edge, device or energy supplier cannot receive energy from any power source. When such a path existed, the objects and relationships in the path are highlighted with red on the transmit-energy layer. In this demonstration, we assume that only these types of objects cannot work without the energy generated in the system. The clause checkDataTransmission(X) aims to discover the edges and devices that cannot transmit any data because of lack of energy. This clause is defined by slightly extending the clause notEnergySupplied(X, Ls). Thus, various clauses can be easily created by reusing existing clauses in accordance with the relationship between layers in order to easily and variously verify the model. Currently, if users want to extend rule clauses, knowledge of Prolog is necessary. Therefore, in future work we plan to establish a method to create a number of beneficial *rule* clauses in line with an architectural model.

A complete program can be generated by combining the *rule* and *fact* clauses mentioned above. The *rule* clauses are used for checking whether a certain object can be reached from another object along the relationships between them on the basis of their behaviors.

IV. A TOOL TO EDIT AND VERIFY MODELS

Fig 1(b) shows a tool to edit and verify models created in TORTE. The tool is implemented in Java and as a software plug-in for the UML modeling editor Astah* [11]. Astah* provides a basic editor for UML diagrams, but adding stereotypes to each element is laborious and error-prone work.

The editor part of the tool provides the following features: (1) matrix-based input forms to the set of architectural objects and to each layer are provided in order to edit object types and

to create relationships including their respective types; (2) six class diagrams representing their respective layers can be automatically generated in order to intuitively confirm the result of inputs; (3) a round trip-engineering function is implemented in order to guarantee the consistency between inputs to the editor and the corresponding class diagrams. For instance, an association and its stereotype represent a relationship and its type, which correspond to a cell in a matrix-based input form. When such an association is removed, the corresponding cell is unchecked automatically.

The verifier part of the tool provides the following features: (1) *fact* clauses in Prolog can be automatically generated from a model in TORTE in order to verify the model; (2) an input form is provided in order to give a *query* to the program; (3) the program is executed by using JIProlog [12], a Prolog implementation in Java, as a checker component; (4) the execution result can be highlighted on the model in order to give feedback to the developer.

V. PRELIMINARY EVALUATION

As a preliminary evaluation, we applied TORTE to an architectural model of an existing farm monitoring system [13], [14]. As a result, we confirmed that the tool can output expected results in some cases. Fig 3 shows an example of the model highlighted. A problem with this model is that the charge controller is not connected to the two DC-DC converters on the transmit-data layer. Elements colored in red in Fig 3 show the edges, devices and energy suppliers whose energy cannot be supplied from any power sources, and the edges and devices that cannot transmit any data because of lack of energy.

VI. CONCLUSION

This paper presented TORTE, a method of modeling an IoT system architecture on the abstraction level of a use case model, and verifying the model with a logic programming language. A support tool provides an editor part by extending an existing UML editor for constructing architectural models, which describe various relationships and relevant objects in

```
1
    /**
 2
    * notEnergySupplied(X, Ls): clauses to identify
 3
    * the paths Ls that energy cannot be transmitted.
 4
    * The paths must include the object X.
 5
    */
   notEnergySupplied(X, Ls) :-
 6
 7
            (object(X, device);
 8
             object(X, edge);
 9
             object(X, energy)),
10
            notEnergySupplied(X, Ls, []).
11
12
   notEnergySupplied(X, Ls, Lp) :-
13
            not(source(energy, X)),
            not(r(transmit_energy, _, X)),
14
15
            Ls = [X|Lp].
16
17
   notEnergySupplied(X, Ls, Lp) :-
18
            not(source(energy, X)),
19
            r(transmit_energy, Y, X),
20
            member(Y, Lp),
21
            Ls = [Y, X | Lp].
22
23
   notEnergySupplied(X, Ls, Lp) :-
24
            not(source(energy, X)),
25
            r(transmit_energy, Y, X),
26
            not(member(Y, Lp)),
27
            notEnergySupplied(Y, Ls, [X|Lp]).
28
   /**
29
    * checkDataTransmission(X): clauses to identify
30
    * whether the object X can work by obtaining energy.
31
    */
32
   checkDataTransmission(X) :-
33
            (object(X, device);
34
             object(X, edge)),
35
            notEnergySupplied(X).
36
37
   notEnergySupplied(X):-
38
            notEnergySupplied(X, Ls).
```

Listing 1. Rule clauses in Prolog (excerpt)

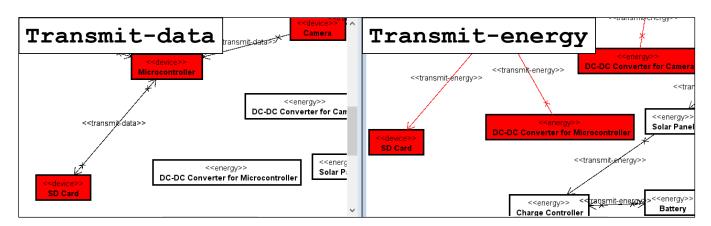


Fig. 3. Highlight of an execution result

IoT systems. The tool also provides a verifier part by using Prolog. As future work, we plan to add beneficial rules to check on the basis of the failure histories of IoT systems. We subsequently evaluate TORTE by applying it to various largescale systems.

ACKNOWLEDGMENT

This work was supported by JSPS KAKENHI Grant Number JP15K00097 and JP17KT0043, and the Telecommunications Advancement Foundation.

REFERENCES

- B. Nuseibeh, "Weaving together requirements and architectures," *Computer*, vol. 34, no. 3, pp. 115–117, 2001.
- [2] J. Cleland-Huang, R. S. Hanmer, S. Supakkul, and M. Mirakhorli, "The twin peaks of requirements and architecture," *IEEE Software*, vol. 30, no. 2, pp. 24–29, 2013.
- [3] E. Yu, P. Giorgini, N. Maiden, J. Mylopoulos, and S. Fickas, *Modeling Strategic Relationships for Process Reengineering*. MIT Press, 2011, pp. 11–152.
- [4] A. van Lamsweerde and L. Willemet, "Inferring declarative requirements specifications from operational scenarios," *IEEE Transactions on Software Engineering*, vol. 24, no. 12, pp. 1089–1114, 1998.
- [5] Object Management Group. (2015) Unified modeling language. (Accessed: 2016-10-04). [Online]. Available: http://www.omg.org/spec/UML/2.5/
- [6] E. S. K. Yu, "Towards modeling and reasoning support for early-phase requirements engineering," in *Proceedings of the 3rd IEEE International Symposium on Requirements Engineering*, ser. RE '97, 1997, pp. 226– 235.
- [7] (2015) Industrial internet reference architecture version 1.7. (Accessed: 2016-07-27). [Online]. Available: http://www.iiconsortium.org/IIRA-1-7-ajs.pdf
- [8] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.
- [9] P. Patel, B. Morin, and S. Chaudhary, "A model-driven development framework for developing sense-compute-control applications," in *Proceedings of the 1st International Workshop on Modern Software Engineering Methods for Industrial Automation*, 2014, pp. 52–61.
- [10] "Information technology Programming languages Prolog Part 1: General core," International Organization for Standardization, Standard, 1995.
- [11] Change Vision, Inc. astah professional. [Online]. Available: http://astah.net/
- [12] U. Chirico. JIProlog. (Accessed: 2016-11-18). [Online]. Available: http://www.jiprolog.com/
- [13] K. Kobayashi and Y. Saito, "Development of hand framing camera for field monitoring," in *Proc. of EFITA-WCCA-CIGR Conference*, 2013.
- [14] K. Kobayashi, Y. Fujikawa, and Y. Saito, "Monorail-based monitoring system for multipoint field observation," in *Proceedings of the 2014 International Workshop on Web Intelligence and Smart Sensing*, 2014, pp. 8:1–8:2.