2016 IEEE International Conference on Software Quality, Reliability and Security

Quantitative Analysis of Variation-Aware Internet of Things Designs using Statistical Model Checking

Siyuan Xu[1], Weikai Miao[1], Thomas Kunz[2], Tongquan Wei[1], Mingsong Chen[1]*
[1]*Shanghai Key Laboratory of Trustworthy Computing, East China Normal University, Shanghai, China, 200063*
[2]*Department of Systems and Computer Engineering, Carleton University, Ottawa, Canada, K1S5B6*
Email: {syxu, wkmiao, mschen}@sei.ecnu.edu.cn, tkunz@sce.carleton.ca, tqwei@cs.ecnu.edu.cn

***Abstract*—Since Internet of Things (IoT) applications are deployed within open physical environments, their executions suffer from a wide spectrum of uncertain factors (e.g., network delay, sensor inputs). Although ThingML is a promising IoT modeling and specification language which enables the fast development of resource-constrained IoT applications, it lacks the capability to model such uncertainties and quantify their effects. Consequently, within uncertain environments the quality and performance of IoT applications generated from ThingML designs cannot be guaranteed. To explore the overall runtime performance variations caused by environmental uncertainties, this paper proposes a quantitative uncertainty evaluation framework for ThingML-based IoT designs. By adopting network of priced timed automata as the model of computation and statistical model checking as the evaluation engine, our approach can model uncertainties caused by external environments as well as support various kinds of performance queries on the extended ThingML designs. Experimental results of two comprehensive case studies demonstrate the efficacy of our approach.**

Keywords—Quantitative Analysis; Internet of Things; Uncertainty Modeling; ThingML

I. Introduction

Due to the capability of bringing billions of online devices in a vast ecosystem, the Internet of Things (IoT) has been identified as a rapidly thriving paradigm of network-of-networks merging both the physical and cyber worlds [1], [2]. It can be expected that IoT will have a high impact on every aspect of daily life. In the near future people can exploit, interact and rely on a wide range of *things* (e.g., sensors, actuators), which will frequently respond to the surrounding environment and cooperate with their peers to reach specified goals.

To enable rapid and efficient application development, model-driven engineering has been widely used in the IoT domain. As a domain-specific modeling language, ThingML [3] enables the model-driven IoT software development, especially for resource constrained IoT applications. ThingML is a combination of architecture models, state machines and an imperative action language. It can be used to describe both software components and communication protocols of IoT applications. Besides modeling, the toolset of ThingML supports automatic code generation from ThingML models to platform-specific implementations (e.g., C, Java). Therefore, the IoT application implementation time and deployment time can be dramatically reduced. By using ThingML, IoT application developers simply focus on the design of high-level models regardless of the heterogeneity of individual *things* and underlying networks.

With the increasing complexity, one major challenge to make the IoT vision a reality is the development of dependable IoT software applications which will be deployed in the real world. IoT devices are deployed in a varying physical environment. There exist various uncertain factors (e.g., network delay, temperature) that cause the execution performance variations of IoT applications. However, ThingML modeling does not take the real-world uncertainty into consideration. Consequently, it is hard to guarantee the QoS (Quality of Service) of IoT applications. Since different high-level ThingML designs within the same physical environment can exhibit very different performance, proper quantitative analysis of ThingML designs within uncertain environments is becoming an important issue to guarantee the QoS of IoT applications.

In order to achieve specified QoS requirements, quantitative analysis of ThingML designs needs to address the two following problems: i) How to accurately model uncertainties caused by external environments in existing ThingML models? ii) How to effectively reason and evaluate the QoS of ThingML models in the presence of such uncertainties? Although probability-based approaches [4], [5] are promising in modeling various kinds of performance variations, few of them can accurately model the concurrent execution of things. Moreover, traditional formal model checking approaches [6] are widely used in checking whether a design can satisfy a given property. However, few of them can reason why QoS cannot be achieved and answer how to improve the QoS. As an alternative, Statistical Model Checking (SMC) [7], [8] is devoted to the quantitative evaluation of system-level designs [9], [10] under variations. By monitoring random simulation runs of

* Mingsong Chen is the corresponding author.

978-1-5090-4127-5/16 $31.00 © 2016 IEEE
DOI 10.1109/QRS.2016.39

systems, SMC can estimate the satisfaction probability of specified properties (i.e., QoS requirements) based on the simulation results using statistical methods (i.e., sequential hypothesis testing or Monte Carlo simulation). Unlike other exhaustive model checking approaches, SMC enables checking a wide spectrum of properties with far less time and memory. Therefore, it is suitable for the quantitative analysis of complex ThingML designs under uncertain environments.

Based on model checker UPPAAL-SMC [11], [12], this paper makes three major contributions as follows:

- We extend the syntax and semantics of the ThingML modeling language, which enables the accurate modeling of performance variations caused by the external uncertain environments.
- We adopt the Network of Priced Timed Automata (NPTA) [8] as the model of computation of our extended ThingML modeling language. Using our proposed mapping rules, extended ThingML designs can be automatically transformed into NPTA models for the quantitative analysis.
- Based on the generated NPTA models, we presented a novel framework that can effectively evaluate the QoS of IoT applications against specified performance queries.

The rest of this paper is organized as follows. After introducing the related work on ThingML and SMC-based evaluation in Section II, Section III provides the background on NPTA and UPPAAL-SMC. Section IV presents the details of our proposed approach. Based on two illustrative case studies, Section V shows that our proposed approach can be effectively applied in the quantitative analysis of extended ThingML designs. Section VI concludes the paper.

II. Related Work

With the rapid advancement in the domain of networking, control, and embedded systems, IoT has established itself as an enabling technology in sensing and controlling the physical world [13]. To facilitate automated analysis and synthesis of IoT systems, model-driven engineering methodologies and platforms [14], [15] are widely used in the top-down design flow of IoT systems. For example, Yang and Pan [16] proposed a novel modeling and analysis approach for real-time IoT designs based on Timing-constraint Petri-nets. The proposed method can be used for the temporal behavior analysis and verification, which can detect and eliminate the timing constraint conflicts from local and global viewpoints. Based on state-machine like semantics, ThingML [17] provides a modeling and code generation platform for resource-constrained IoT applications. Although more and more model-driven approaches are investigated to facilitate the construction of IoT systems, most of them focus on the functional correctness of the design. Few of them consider the performance variations caused by the uncertain environments at the level of design models. Since IoT systems are deployed in an uncertain physical environment, if the variations caused by the environment cannot be reflected at the system-level models and the impact of such variations cannot be estimated, the QoS of the derived target systems cannot be guaranteed.

Unlike traditional model checking approaches [6] which can only answer *yes* or *no* for a given design and safety-properties, quantitative analysis using SMC is a process of measuring the probability of a whole execution satisfying certain temporal properties [18]. Due to the efficacy in evaluating performance-related metrics and user-friendly interface, UPPAAL-SMC [11], [12] is becoming one of the mostly used tools for the quantitative analysis of system-level designs. For example, David et al. [11] extended the semantics of UPPAAL-SMC in order to facilitate the modeling and evaluation of hybrid systems in various domains (e.g., biology systems, energy-aware buildings). In [19], Du et al. adopted UPPAAL-SMC to quantitatively evaluate project schedules. Their approach supports various performance queries, which can be used to compare different aspects of project schedules. Based on UPPAAL-SMC, Chen et al. [9] proposed a novel framework that supports the modeling and evaluation of resource allocation strategies in Cloud computing. By using their approach, Software as a Service(SaaS) providers can not only filter out inferior resource allocation solutions under variations in an efficient manner, but also can tune requirement parameters to achieve better profit. Similarly, in [10], Chen et al. introduced an UPPAAL-SMC-based approach for the quantitative evaluation of task allocation and scheduling in the MPSoC domain considering both time and power variations.

Although there are dozens of model-driven approaches that can benefit IoT design, most of them focus on the functional issues. To the best of our knowledge, there are no existing methodologies and tools that investigate the impact of uncertain environments on the IoT designs and allow the optimization of IoT design performance under variations. Our proposed approach is the first attempt that not only supports the variation modeling of ThingML, but also allows the quantitative evaluation and comparison of variation-aware IoT designs from a performance perspective.

III. Preliminary Knowledge of SMC

Unlike traditional timed automata whose clock rate is always 1, the clocks in a Priced Timed Automaton (PTA) can evolve with different clock rates. Let X be a clock set. A *clock valuation* is a mapping function $v : X \rightarrow R_{>=0}$ from the set of clocks X to the set of non-negative reals $R_{>=0}$. Let $v_0(c) = 0$ for all $c \in X$, where v_0 indicates the initial valuation. Let $\mathcal{B}(X)$ be a set of finite conjunctions of clock expressions in the form of $x \sim k$ or $x - y \sim k$, where $x, y \in X$, $k \in R$, and $\sim \in \{<, \leq, ==, >, \geq\}$. Assuming $g \in \mathcal{B}(X)$, $v(X) \models g$ indicates that valuation $v(X)$ satisfies the constraint g. To simplify the formal modeling, we focus

on the key components of PTAs while skipping the richer flavors of PTAs supported by UPPAAL-SMC, e.g., *urgent locations* [7]. The PTA [7] can be formally defined as follows.

Definition 1. *A PTA is a 8-tuple* $(L, l_0, \Sigma, X, \tau, F, I, E)$ *where:*

- L *is a finite set of locations.*
- l_0 *denotes the initial location.*
- Σ *is a finite set of exclusive input and output actions.*
- X *is a finite set of clocks.*
- τ *is the system clock which will not be reset.*
- $F : L \to f^X$ *assigns a clock rate vector to each location, where* $f(x)$ *specifies the rate of clock* x.
- $I : L \to \mathcal{B}(X)$ *assigns an invariant to each location.*
- $E \subseteq L \times \mathcal{B}(X) \times \Sigma \times 2^X \times L$ *denotes the finite set of transition edges, where* $\mathcal{B}(X)$ *indicates transition guard set and* 2^X *denotes the set of reset clocks.*

Since PTAs can communicate with each other through broadcast channels and shared variables [8], a set of correlated PTAs can be composed as a network, named Network of Priced Timed Automata (NPTA). Assume that $(l, v) \in L \times R^X_{>=0}$ is a state of an NPTA where $v \models I(l)$. Let $v[Y]$ be the reset operation on the clock set Y. In other words, if $x \in Y$, $v(x)$ will be reset to 0, otherwise the value of $v(x)$ does not change. The semantics of NPTA is mainly based on the following two kinds of transitions: i) A transition in the form of $(l, v) \xrightarrow{a} (l', v')$ is a *discrete transition* iff there is a transition (l, g, a, Y, l') in current state, where $v \models g$ and $v' = v[Y]$. ii) A transition in the form of $(l, v) \xrightarrow{d} (l, v')$ is a *delay transition* iff $v' = v + \int_{v(\tau)}^{v(\tau)+d} F(l)\, d\tau$, where $v(\tau)$ indicates the system time of entering state (l, v) and both $v \models I(l)$ and $v' \models I(l)$.

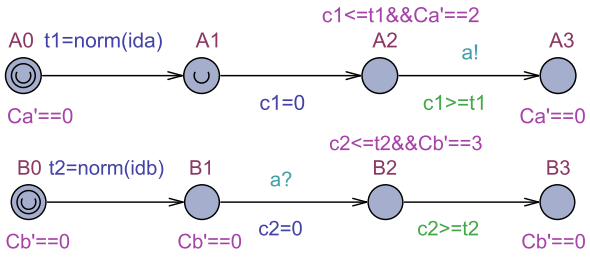

Figure 1. An NPTA (A | B)

Figure 1 shows an NPTA with two PTAs A (id=ida) and B (id=idb). The locations marked with "U" are *urgent locations*. All the urgent locations freeze time, *i.e.* time is not allowed to pass when a PTA is in an urgent location [8]. Each PTA has four locations and two clocks (e.g., Ca and $c1$ for A), where $c1$ and $c2$ are system clocks. Note that in different locations, clocks can evolve with different rates (i.e., unit price) determined by the specific clock rate vector. By default, the rate of a clock is 1. To change the rate of a clock in some location, we need to specify the new rate to the primed version of the clock. For example, $Ca' == 2$ in location $A2$ indicates the clock rate in $A2$ is 2. Since the clock rate can be considered as a specific kind of unit price, the value of the clock can be considered as the corresponding cost. If we stay there for 2 seconds, then the cost of staying there for 2 seconds is 4. In this example, assume that the values of two variables $t1$ and $t2$ follow the normal distribution $N(2,1^2)$ and $N(4,2^2)$, respectively. The action $t1 = norm(ida)$ on the outgoing transition edge of $A0$ assigns $t1$ with a random value following the normal distribution $N(2,1^2)$, which indicates the mean value is 2 and its standard deviation is 1. Since the invariant of $A2$ contains $c1 \leq t1$ and the guard on the outgoing edge of $A2$ is $c1 \geq t1$, PTA A will stay at location $A2$ for a time of $t1$. In this way, if PTA A runs for large number of times, the sojourn time at location $A2$ will follow the specified distribution. In this example, the PTAs A and B are synchronized by two complementary actions "a!" and "a?" via an *urgent broadcast channel* a.

$$
\begin{aligned}
&((A_0, B_0), [c_1 = 0, c_2 = 0, C_a = 0, C_b = 0]) \xrightarrow{0} \\
&((A_1, B_1), [c_1 = 0, c_2 = 0, C_a = 0, C_b = 0]) \xrightarrow{0} \\
&((A_2, B_1), [c_1 = 0, c_2 = 0, C_a = 0, C_b = 0]) \xrightarrow{2.3} \\
&((A_2, B_1), [c_1 = 2.3, c_2 = 2.3, C_a = 4.6, C_b = 0]) \xrightarrow{a} \\
&((A_3, B_2), [c_1 = 2.3, c_2 = 0, C_a = 4.6, C_b = 0]) \xrightarrow{4.1} \\
&((A_3, B_3), [c_1 = 6.4, c_2 = 4.1, C_a = 4.6, C_b = 12.3]) \xrightarrow{\cdots}
\end{aligned}
$$

The above example shows a random feasible execution of NPTA $(A|B)$. It shows that the composite location (A_3, B_3) can be reachable within 6.4 time units with a total cost of $C_a = 4.6$, $C_b = 12.3$. To enable the evaluation, UPPAAL-SMC will generate a large quantity of random runs to estimate the overall performance of the NPTA. Since the execution of $A2$ and $B2$ of NPTA $A|B$ is independent and both of them start from time 0, the arriving time at composite location $(A3, B3)$ follows the distribution $N(6, 1^2 + 2^2)$. Note that since UPPAAL-SMC supports the programming of distributions based on the built-in function *random*() and C-like programming constructs for NPTA, by using the above variation modeling template, arbitrarily complex stochastic behaviors can be modeled.

Based on the statistical approaches, SMC enables the quantitative evaluation of NPTA-based designs using cost-constrained temporal logic [11] based properties in the form of $Pr[cost <= bound](<> expr)$. Here, *bound* is a constant value, the bound information $[cost <= bound]$ restricts the maximum value of *cost*. By default, the *cost* denotes the system clock if it is not specified explicitly. In the property, the expression $<> expr$ asserts that the predicate *expr* will hold eventually. According to the specified *probability of false negatives* (i.e., α) and *probability uncertainty* (i.e., ε), UPPAAL-SMC will generate a set of stochastic runs which are terminated when either $cost <= bound$ or $<> expr$

holds. By monitoring the results of randomly generated runs, the probability range of each property (i.e., $[p-\varepsilon, p+\varepsilon]$) with a specified confidence (i.e., $1-\alpha$) will be reported, where p indicates the success ratio of the given property.

IV. Our approach

Figure 2 shows an overview of our approach. Unlike traditional qualitative analysis approaches, within an uncertain environment ThingML designers would like to query "how much a design requirement can be fulfilled by a specific ThingML design?" To enable such quantitative evaluation of ThingML designs with variation information, our approach has two inputs: i) design requirements indicating user-expected performance metrics, and ii) extended ThingML designs coupled with the variation information (e.g., network delay, sensor inputs). In our approach, the design requirements are translated into performance queries in the form of cost-constrained temporal logic based properties. The extended ThingML designs and variation information are firstly parsed and saved as ThingML meta models. Then, based on our proposed transformation rules, the meta models are transformed into NPTA models. By using the model checker UPPAAL-SMC [8], our approach can conduct the quantitative analysis and report the evaluation results. The following subsections will describe the details.

A. Modeling of Variation-Aware ThingML

A typical ThingML design usually consists of three parts: i) *state charts* that model the behaviors of individual things and communications between things; ii) *functions* that support the programming of functional procedures using an imperative action language; and iii) *architecture models* that elaborate the interconnection and hierarchy of things. Based on the above modeling constructs, various IoT functionality can be modeled. However, due to the lack of performance modeling mechanisms in the current ThingML version, it is hard to verify the performance metrics of corresponding IoT implementations. The situation becomes even worse when IoT applications are deployed within an uncertain environment. For example, IoT things deployed within uncertain environments often suffer from the *input uncertainty*. This kind of uncertainty may come from a variety of resources, including sensors and human activities. For example, the input value of a temperature sensor varies along with the unpredictable weather. As another example, *communication uncertainty* strongly affects the system performance during the data transmission among *things*. Usually, it is hard to guarantee the data transmission time due to the lack of predictability for network congestion, payload size of transmitted messages, routing failures, etc.

The current version of ThingML assumes that all the *things* are working within an ideal environment. For example, in ThingML the event-based communication is assumed to be instantaneous, which cannot reflect real interconnection dynamics between things in practice. Since the designers have no knowledge about the impacts of performance variations in system-level design, the QoS of IoT implementations cannot be optimized or guaranteed. To enable the performance evaluation of ThingML designs under variations in an early stage, this paper extends the syntax and semantics of ThingML to support the modeling of performance variations.

Definition 2. *A variation-aware Thing design is a 12-tuple* $(S, s_0, T, A, E, G, M_E, M_T, Cond, P, \Sigma, M_\Sigma)$*, where:*

- S *is a finite set of states.*
- s_0 *is the initial state.*
- $T \subseteq S \times S$ *denotes the finite transition set.*
- A *is a finite set of actions.*
- E *is a finite set of events.*
- G *is a finite set of guard conditions.*
- $M_E : E \to T$ *maps events to its corresponding transitions.*
- $M_T : T \to DIST$ *assigns each transition with a distribution of action execution time on that transition, where* $DIST$ *is a set of distributions.*
- $Cond : G \to T$ *maps guard conditions to their corresponding transitions.*
- P *is a finite set of ports, where a port supports multiple message send and message receive operations.*
- Σ *is a finite set of Thing properties, where each property specifies a variable with specific data type and initial value.*
- $M_\Sigma : \Sigma \to DIST$ *assigns each property with a value distribution, where* $DIST$ *is a set of distributions.*

Definition 2 gives the formal definition of variation-aware things. A *thing* instance is created to accomplish some task by itself or collaborate with other things to accomplish a more complex task. By adopting the syntax and semantics similar to state machines, a *thing* can sense the external environment and save its value in variables defined as properties, conduct the calculation in actions using an imperative language provided by ThingML, make the control decision based on the guards of the underlying state machine (defined as a *"statechart"*), and communicate with other things via ports.

As the underlying model of computation, a statechart mainly consists of *states*, *transitions*, *actions*, *guards* and *events*. The states indicate different statuses of *things*. The transitions specify the updates between states when specific events happen and corresponding guards hold. The complementary events (e.g., "e!" and "e?") on the paired transitions of different statecharts are used for the purpose of synchronization. They are used to denote the communication between *things*. In ThingML, actions can be statements for variable assignment or functional calls. When a transition is triggered, some calculation will be conducted to reflect the effect of corresponding transition actions. Originally, the ThingML statechart does not support the modeling of

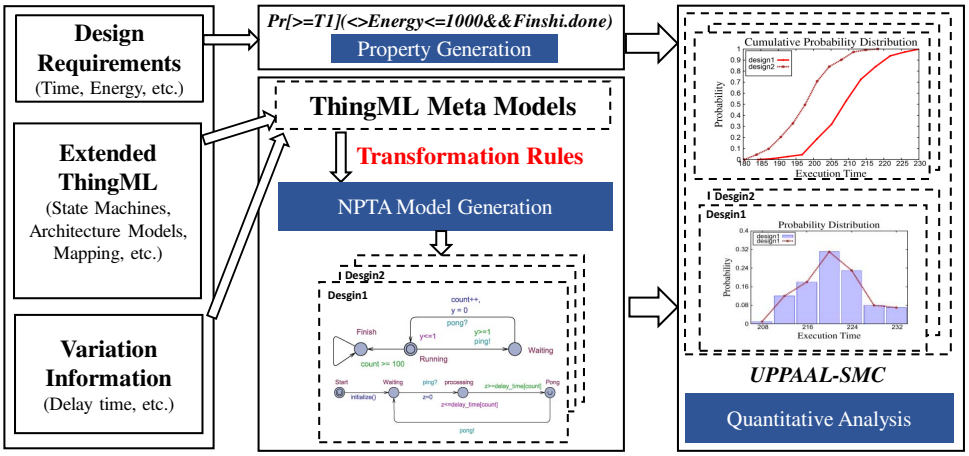

Figure 2. The workflow of our approach

timing behaviors of IoT systems. It is assumed that all the transitions are triggered instantaneously and the action calculation has no delay. To consider the varying environments and enable the modeling of timing behaviors, we extend ThingML statecharts for the purpose of quantitative analysis. To model the uncertain execution time of combined actions on a transition, we assign the transition with an execution time distribution for all actions. By default, the execution time of a transition is 0 if there is no action time distribution associated with that transition. In order to hold values for a specific state and its inputs/outputs, *properties* are introduced to define and initialize corresponding variables and constants (with a prefix "*readonly*"). Note that since properties can be used to sense the uncertain environments or human activities, the value of such properties cannot be pre-determined. To model the value of such properties during the stochastic execution, we assign each property with a value distribution. The *port* plays an important role in the communication between *things*. As an interface for communication, a port allows for grouping multiple messages together.

Definition 3. *A variation-aware ThingML design is a 4-tuple* $(Th, C, Arch, M_C)$*, where:*

- *Th is a finite set of correlated things.*
- *C is a finite set of communication channels (messages) between things.*
- *Arch is an architecture model consisting of a set of rules, which instantiates the things in Th and specifies the connectors (channels) between ports of things.*
- $M_C : C \rightarrow DIST$ *assigns each message with a distribution of network transmission time, where DIST is a set of user-defined distributions.*

Definition 3 gives the formal definition of a variation aware ThingML design. An IoT application involves a large set of collaborating *things*. All these things communicate with each other via messages on top of channels, which are bidirectional links that connect to different Thing ports. To elaborate an executable ThingML application, we need to define an architecture model, which instantiates all the things and establishes physical connection between things using *connectors*. Since there exist various uncertain factors that can affect the communication between things, to model the transmission variation, we assume that the message latency/delay time follows some distribution. Therefore, we assign a transmission time distribution to each message (channel). To enable quantitative performance analysis of ThingML designs, our approach supports the variation modeling for properties, action execution time and message transmission time. Note that in the same way other kinds of variations can be easily integrated into ThingML.

B. NPTA Model Generation

Listing 1 shows an extended ThingML design of *P*ingPong example (see details in Section V-A). The design includes two state machines (i.e, a *Client* and a *Server*) and an architecture model. In this example, the Thing fragment *PingPongMsgs* defines the message information. Since both messages *ping* and *pong* are used in both the client and server, with the keyword *includes* the fragment *PingPongMsgs* is incorporated into both things *Client* and *Server*. The Thing *Client* has two properties, i.e., a constant *count_max* which indicates the limit (i.e., 100) of *PingPong* interactions between the two things, and a variable *count* which denotes the index of the current *PingPong* interaction. The port definition *ping_service* indicates that the port only allows the messages *ping* and *pong*. The function *counterInc()* is an action which increases the property *count* by 1. The *statechart* specifies the underlying state machine for the corresponding Thing. For the *Client*, we assume that the *ping* message is triggered instantaneously and the action *counterInc()* lasts for 1 second. In this example, we assign the transition from *Waiting* to *Running* with a

time distribution denoted by *"follow constant(1.0)"* in its comment. Here, *constant(1.0)* means a constant of 1 second. The architecture model at the bottom shows the elaboration of the instantiated things. In this example, we assume that the transmission time of the *pong* message follows the normal distribution $N(1.0, 0.2^2)$. Here, connectors in the architecture model specify the channels between ports. Therefore, in the comment of the second connector in the example, the transmission time distribution is denoted by *"follow Normal(1.0,0.2)"*. Note that in this example, we do not consider the distributions for input variations. If needed, such distribution information can be denoted in the comments using the keywords *follow*.

```
1  thing fragment PingPongMsgs{
2      message ping();
3      message pong();
4  }
5  thing Client includes PingPongMsgs {
6      readonly property  count_max : Integer = 99
7      property count: Integer = 0
8      provided port ping_service {
9      sends pong
10     receives ping
11     }
12     function counterInc():Void do
13     count = count + 1
14     end
15     statechart PingClientMachine init Running {
16     state Waiting {
17     transition -> Running //follow constant(1.0)
18     event ping_service?pong
19     action counterInc()
20     }
21     state Running{
22     transition -> Finish
23     guard count > count_max
24     transition -> Waiting
25     action ping_service!ping
26     }
27 }
28 thing Server includes PingPongMsgs{
29     provided port pong_service {
30     receives pong
31     sends ping
32     }
33     statechart PingServerMachine init Waiting{
34     state Waiting{
35     transition -> Pong
36     event ping_service?ping
37     }
38     state Pong{
39     transition -> Waiting
40     action ping_service!pong
41     }
42 }
43 //architecture model
44 configuration PingPongCfg
45 @arduino_stdout "Serial" {
46     instance thing1: Client
47     instance thing2: Server
48     connector thing1.ping_service
49     => thing2.ping_service
50     connector thing2.pong_service
51     => thing1.pong_service //follow Normal(1.0,0.2)
52 }
```

Listing 1. An extended ThingML design for *P*ingPong example

To parse both structure and variation information from extended ThingML designs, ThingML toolset provides a set of EMOF (Eclipse Model Driven Framework)-based APIs, which can extract the abstract syntax tree of ThingML based on meta models. Listing 2 shows a snippet of the hierarchical meta model definition for ThingML designs. In the definition, *List* denotes the data structure list, whereas *EList* is an extension of *List* which supports the movement of its elements. For example, the state machine *Client* in Listing 1 has a list of two states (*Running* and *Waiting*) and has a list of two properties (*count* and *count_max*). To enable the parsing of variation information saved in comments, we slightly modified the existing EMOF-based APIs. According to the ThingML meta models, the variation information can be captured and saved as string-based annotations. For example, when the parsing process reaches line 17 of Listing 1, the comment "*follow constant(1.0)*" will be parsed together with the transition information and the action time distribution will be saved as an annotation of the transition.

```
1  Thing{
2      String name;
3      EList<Property> properties;
4      Elist<Function> functions;
5      EList<StateMachine> stateMachines;
6      ...
7  }
8  StateMachine{
9      Sting Name;
10     List<State> states;
11     List<Property> properties;
12     ...
13 }
14 State{
15     String Name;
16     EList<Transition> outgoingTransitions;
17     EList<Transition> ingoingTransitions;
18     ...
19 }
20 Transition{
21     State target;
22     EList<Event> event;
23     Action action;
24     String Annotations;
25     ...
26 }
```

Listing 2. A snippet of ThingML meta model

To enable automated quantitative analysis, all the extracted information from ThingML designs needs to be transformed into NPTA models. Based on the meta model, we classify the ThingML constructs into four categories, i.e., *data structure*, *state machine*, *architecture model* and *procedure call*. The *data structure* mainly deals with the variable declaration and initialization. While the *state machine* outlines the behavior of a single thing, the *architecture model* elaborates the network of collaborating things. The *procedure call* programs action details or operations in the form functions. Table I presents an overview of the mappings of these four kinds of constructs between ThingML designs and NPTA models. For example, in the category *architecture model*, the connectors extended with transmission time variation information will be converted into its NPTA counterparts in the form of actions, location invariants and transition guards.

In our approach, the target NPTA model consists of two parts: i) front-end models which are used to describe the behaviors of PTAs, and ii) back-end configurations which are used to declare necessary data structures (e.g., variables, channels) and functions to support the stochastic execution of the NPTA. During the transformation, we assign each *thing* with a front-end model and a local back-end configuration. Besides local configurations of PTAs, the NPTA itself has two global back-end configurations, i.e., *system declaration* which is used to instantiate PTAs and elaborate the NPTA, and *global declaration* which is used to define global data structures and functions shared by all PTAs.

Table I
MAPPINGS OF CONSTRUCTS BETWEEN *ThingML* AND *NPTA*

Category	ThingML Constructs	NPTA Constructs
Data Structure	Property	Variable
	Message	Channel Declaration
State Machine	State	Location
	Extended Transition	Transition, Action, Invariant, Guard
	Guard	Guard
	Event	Channel?
	Action	Channel!, Update
Architecture Model	Port	-
	Extended Connector	Action, Invariant, Guard
Procedure Call	Function	Function
	Operator	Function

Back-end Configuration Generation: Listing 3 shows the skeleton of back-end configurations of the *PingPong* example. To save the space, we put the *system declaration*, *global declaration* and local configuration for PTAs together within a listing. All these configurations are generated from the ThingML design shown in Listing 1.

The first part of Listing 3 shows the details of *system declaration* configuration. From the ThingML architecture model, we can figure out the details of things (i.e., name and type). For each thing, we will instantiate one instance of the corresponding front-end model. With the keyword *system*, we can construct the NPTA of all the interconnected things.

The second part of Listing 3 presents the *global declaration* configuration for the *PingPong* example. To enable the communication between things, ThingML adopts a *message*-based approach, where a message can be considered as an encapsulation of data values. In NPTA, PTAs communicate each other through channels and shared variables, where channels are used for the synchronization and share variables are used to hold data values. Therefore, during the transformation, for each message in ThingML we need to create an urgent broadcast channel and a data structure of multiple variables to hold the data values. For example, the ThingML statement *"message pong();"* (line 3 in Listing 1) is converted to its NPTA counterpart *"urgent broadcast channel pong"*. Since the *pong* message has no associated data, we do not create the corresponding variables in the back-end configuration. Since our approach supports the variation-aware IoT designs, to enable stochastic modeling the *global declaration* configuration includes a library of various commonly used distributions.

```
1  //system declarations
2  thing1 = Client();
3  thing2 = Server();
4  system thing1, thing2;
5  //-------------------------------------------------
6  //global declarations & distribution function lib.
7  urgent broadcast channel ping;
8  urgent broadcast channel pong;
9  double Normal(double mean, double deviation){
10    //  BoxMuller method
11    ...
12 }
13 double Uniform(double min, double max){
14    ...
15 }
16 ...
17 //-------------------------------------------------
18 //local configuration for thing1.
19 const int count_max = 99;
20 clock Cl_clk;
21 int count;
22 double t1;
23 void initialize(){
24     count = 0;
25 }
26 void counterInc(){
27     count = count + 1;
28 }
29 //-------------------------------------------------
30 //local configuration for thing2.
31 ...
```

Listing 3. Back-end configuration of PingPong

For each thing, we create one local back-end configuration, which defines the data structures and functions required by the corresponding PTA. Different from the properties of UPPAAL-SMC (see details in Section IV-C) which are used for performance checking, *property* is an important construct in ThingML which defines variables and constants. For example, *"readonly property count_max : Integer = 99"* is a declaration in the *PingPong* example, which indicates the total number of *PingPong* interactions. Due to the keyword *readonly*, the statement will be transformed into a constant declaration "*const int count_max = 99*" in the back-end configuration (line 19 in Listing 3). Without the keyword *readonly*, the ThingML properties will be transformed into variables in local configurations, and its value will be initialized in the local function *initialize()* of the *thing*. For example, the ThingML property *count* (line 7 in Listing 1) is translated as a variable in the local back-end configuration (line 21 in Listing 3) whose value should be initialized. Note that each PTA has its own local function *initialize()* to conduct the initialization of local variables and clocks. ThingML supports two kinds of functions which are specified using the keywords *function* or *operator*. Similar to ThingML functions, ThingML operators can be used as actions of things. The only difference is that the return values of operators should be of boolean type. Since NPTA

functions can be used as actions to update the input/output variables, they can be directly derived from their ThingML counterparts, though the format is slightly different. For example, the function block in lines 12-14 of Listing 1 can be translated into the a C-like function (see lines 26-28 in Listing 3) defined in the local configuration of *thing1*.

Front-end Model Generation: In our approach, a front-end PTA model is used to describe the stochastic behavior of a *thing*. The structure of a front-end models mainly comes from statechart definitions in ThingML designs. To illustrate the our front-end model generation, Figure 3 shows two front-end models (i.e., client and server) for the *PingPong* example generated from Listing 1.

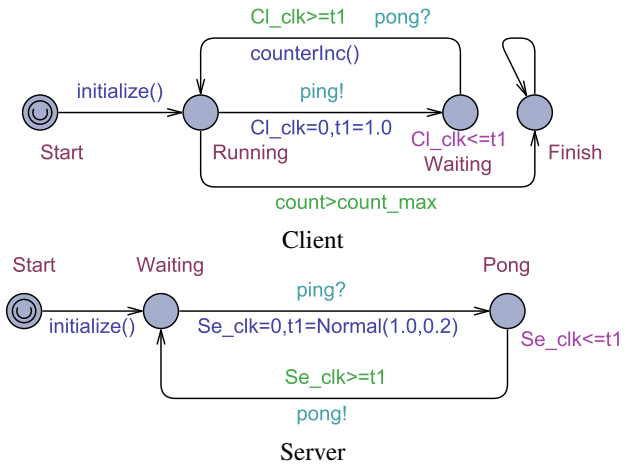

Figure 3. NPTA of *PingPong* example

As shown in Table 1, during the transformation, all the states, transitions, and guards of a ThingML statechart can be directly mapped to their PTA counterparts. To enable the initialization of local variables, one extra location named *start* is introduced for each PTA. The *start* is an urgent location that has no delay. The function *initialize()* on the outgoing edge of *start* can be considered as an action which is used to initialize values of variables and clocks. In the *PingPong* example, we only consider two kinds of variations, i.e., the action execution time variation for *counterInc()* and the transmission time variation for message *pong*. For the extended transition *"transition→Running"* with variation information *"follow constant(1.0)"* in line 17 of Listing 1, to model the stochastic execution, the transformation involves three NPTA constructs: i) the action on the incoming transition of the location, ii) the invariant of the location, and iii) the guard of the outgoing transition of the location. In this example, *Cl_clock* and $t1$ are temporary clocks or variables defined in the local back-end configuration of PTA *Client*. We reset the clock *Cl_clock* and initialize the value of variable $t1$ on the incoming transition of location *Waiting*. Here, $t1$ indicates the stochastic execution time of the action. We set the location invariant of *Waiting* to $Cl\_clk \leq t1$, and set the outgoing transition guard of *Waiting* to $Cl\_clk \geq t1$. Since *const(1.0)* denotes a constant of 1.0, there is no need to explicitly call some function in the distribution function library. Similar to the template introduced in Figure 1, all these three mapping settings can guarantee the stochastic action execution time follows the given distribution.

For both ThingML and NPTA, the communication between things or PTAs are based on synchronization of events. In ThingML, the communication is conducted using messages. The sender of the message firstly initiates a message using action *msg!*. In the same time if there is a receiver waiting for the message using the event *msg?*, the communication will succeed. Note that the sending action of a ThingML message can be considered as a special event. For example, the statement "*action ping_service!ping*" is an event initiated by the *Client*, and the statement "*event ping_service?ping*" is the matching event triggered by the *Server*. In NPTA, the communication between PTA is based on the channels. Assume that *ch* is a channel. The complementary events *ch*! and *ch*? denote the sending and receiving events, respectively. In our approach, the channels are all globally defined in the *global declaration* configuration. Therefore, the transformation from message-based communication to the channel-based communication is quite straightforward in front-end models. For example, the ThingML *ping* message events aforementioned can be transformed as complementary events *ping*! and *pong*? in corresponding PTAs. During the transformation, since a ThingML *port* has no concrete meaning,we do not specify any rules for its transformation. In our approach, we use the connectors defined in ThingML architecture models to specify the data transmission time variation. For example, the lines 50-51 in Listing 1 assigns a normal distribution *Normal(1.0,0.2)* to the corresponding *pong* message. To enable the stochastic modeling, we create a clock *Se_clk* and a variable *t1* in the local configuration of *Server*. By using the same transformation rules for extended transitions, we can specify the data transmission time variation on the generated NPTA model.

C. Property Generation for Quantitative Analysis

The cost and responsiveness are two major issues in ThingML design under variations. During the system-level design, ThingML designers would like to ask the questions like ''what is the probability that a specific function scenario can happen within a given time limit?" or ''what is the probability that the system consumes more resources than required within a given time limit?". To evaluate such QoS requirements, our approach mainly investigates the following two kinds of queries for IoT designs: i) *status query* in the form of property $Pr[<= T](<> instance.state)$ which evaluates whether the *state* of an IoT instance can be reached within a time limit T; and ii) *cost query* in the form of

property $Pr[<=T](<>cost>=K)$ which evaluates whether the *cost* (i.e., the value of clocks or variables) can exceed K within a time limit T. Note that these queries are by no means the "golden" queries rather they are representative ones for common purposes. ThingML designers can improve or modify these queries for their special applications.

```
requirement PingPongReq{
    squery Q1{
        bound: <= 500
        thing: thing1
        state: Finish
    }
    cquery Q2{
        bound:    <= 500
        resource: thing1.count
        cost:     80
    }
}
```

Listing 4. A design requirement example

Since ThingML does not support the description of performance queries, we adopt a separate file to specify performance queries for IoT designs. Note that the requirements are parsed and transformed by our own transformation tool. Listing 4 shows a design requirement example with two performance queries. In this example, the keywords *squery* and *cquery* denote the status query and cost query, respectively. The keywords *bound*, *thing* and *state* within status queries specify the time bound, device and expected state, respectively. Based on the provided performance parameters, the query *Q1* will be converted into a property $Pr[<=500](<>thing1.Finish)$ to check whether the *Client* in the *PingPong* example can finish within 500 time units. Similarly, the cost query *Q2* will be converted into a property $Pr[<=500](<>thing1.count>=80)$ to check whether the number of *PingPong* interactions can be greater than or equal to 80 within 500 time units.

V. Case Study

To show the efficacy of our approach, this section presents the experimental results of two ThingML designs which are collected from the HEADS projects. The first design is the *PingPong* [20] which describes the interactions of things within a client-server architecture. In this example, we focus on the impact of uncertain network delay during the data transmission. The second design is a *heating control system* [21] where we put emphasis on the energy consumption variations caused by ambient temperature and human activities. To enable quantitative evaluation of variation-aware ThingML designs, we developed a tool chain that integrates ThingML, UPPAAL-SMC (version 4.1.19) and our NPTA model generator (implemented in Java). In the experiments, we set both ε and α of UPPAAL-SMC to 0.02. All the experiment results were obtained on a desktop with 3.10 GHz Intel i5 CPU and 8 GB RAM.

A. Case Study 1 - PingPong Design

The *P*ingPong design was used through Section IV to illustrate the workflow of our approach. The design consists of two things, i.e., a client and a server. The client periodically sends a *ping* message to the server. Once receiving the message, the server will send *pong* as a reply. Figure 3 shows the NPTA generated from our extended *P*ingPong design from [20], named *design1*. To enable the performance comparison under variations, we slightly modified the original version and got an alternative design for the *PingPong* example whose NPTA is shown in Figure 4, named *design2*. In both designs, we assume that each *ping* action is triggered instantly followed by a pause of 1 second, while the *pong* actions suffer from the network delay. In *design1*, we assume that the transmission time of *Pong* messages follows the normal distribution $N(1.0, 0.2^2)$. Since *counterInc()* increases the counter by 1 in each iteration, there are a total of 100 iterations before reaching the location *Finish*. Unlike *design1*, *design2* merges two consecutive *pong* replies into one combined reply. In *design2*, we assume that the transmission time of each combined reply follows the normal distribution $N(2.0, 0.36^2)$. Since the counter is increased by 2 in each iteration, *design2* needs 50 iterations to reach the location *Finish*.

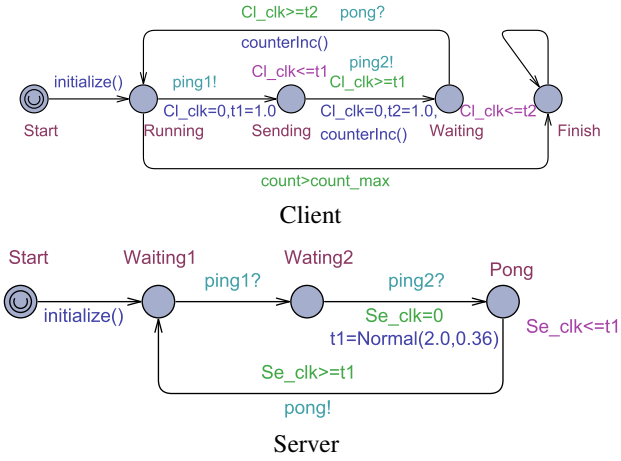

Figure 4. NPTA of *design2*

Without considering the transmission variations for *pong*, both *design1* and *design2* should have the same performance, since the average mean time for *ping* and *pong* messages are the same. However, within uncertain environments, different designs show different performance. Since both *design1* and *design2* only consider the variation of transmission time, we use the status query to evaluate the performance of different designs. Assume that the designers want to check *"what is the probability that a client can perform 100 PingPong interactions with the server within 500 unit times?"*. Based on the user-defined queries in the ThingML configuration as shown in Listing 4, our tool can automatically generate the

status query $Pr[<=500](<>thing1.Finish)$.

Figure 5 presents the Cumulative Probability Distribution (CPD) results for the query on the two different designs. The x-axis denotes the time limit, and the y-axis indicates the success rate of the given query with variations. By running 113 simulation runs, both queries can obtain a success ratio within range [0.96,1] with a confidence of 98%. When checking use UPPAAL-SMC, both queries consume around 30 seconds for the quantitative analysis. From this figure, we can observe that both designs can finish 100 *pingpong* interactions within the given time limit (i.e., 500 seconds). However, *design2* outperforms *design1* since *design2* needs less execution time to achieve the same success ratio. To achieve the highest success ratio, *design2* only needs an execution time of 288 seconds for the *pingpong* interactions while *design1* needs 309 seconds.

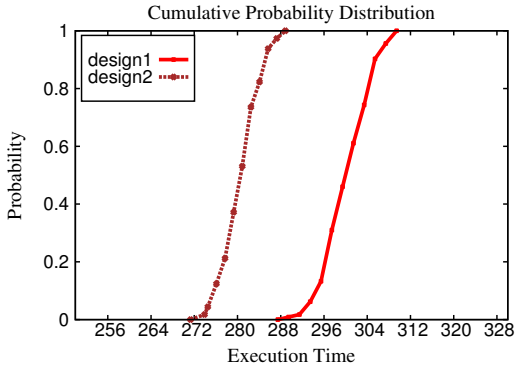

Figure 5. Performance comparison between *design1* and *design2*

To demonstrate that high-level ThingML designs with better evaluation results will lead to corresponding IoT implementations with better performance, we checked the performance of IoT implementations (in C++) which are automatically generated from the two ThingML designs. It is important to note that ThingML only uses the message-based communication, which cannot fully reflect the performance in practice. To consider more details of the real network environment, we adopted the network simulation platform OMNeT++ [22] and deployed the two IoT implementations on it. To simplify the comparison, we used the default routing protocol provided by OMNeT++. Since we focused on the transmission time variation of *pong* messages, we employed the same transmission time distributions for the OMNeT++ simulation as the one used in high-level ThingML designs. Similar to UPPAAL-SMC, we conducted 113 stochastic simulations for the two alternative *PingPong* designs, respectively. For each simulation, we recorded the overall transmission time of 100 *PingPong* interactions (or 50 combined interactions).

Figure 6 and Figure 7 show the probability distributions of the transmission time for *design1* and *design2* simulated using OMNeT++, respectively. From these two figures, we

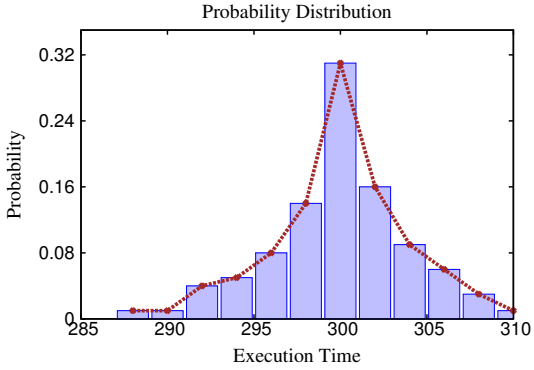

Figure 6. Simulation results of *design1* using OMNeT++

can find that *design2* has a better performance than *design1*, since it require less average transmission time. Moreover, compared with the OMNeT++ based approach which costs a simulation time of around 3 minutes, our UPPAAL-SMC based approach only needs a simulation time of around 30 seconds. In other words, our approach enables the quick exploration of optimized variation-aware IoT designs in an early stage with much fewer evaluation efforts.

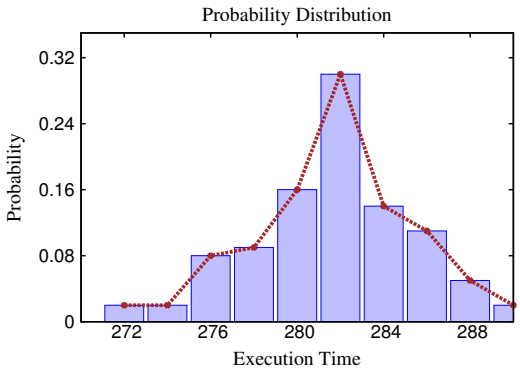

Figure 7. Simulation results of *design2* using OMNeT++

B. Case Study 2 - Heating Control System

As a typical IoT system, the heating control system [21] consists of multiple kinds of interconnected things including timers, indoor temperature sensors, outdoor temperature sensors, person detectors, alert components, and air conditioners. To simplify the modeling, the ThingML design contains only one instance for each category of things. Due to the space limit, we do not present the details of ThingML design and corresponding generated NPTA models for the heating control system. Table II only presents an overview of the heating control system. In the table, the first column denotes the PTA instance. The second and third columns indicates the number of locations and transitions of the design, respectively. The last column describes the function of each PTA. In this case study, we focused on the energy

Table II
AN OVERVIEW OF HEATING CONTROL SYSTEM

PTAs	# of Locations	# of Transitions	Functionality
Timer	3	4	Trigger timing events periodically
Person Detector	2	2	Detect the presence of people in a room
Alarm Component	4	5	Ring the alarm when meeting some condition
Indoor Temperature Sensor	3	3	Sense indoor temperature
Outdoor Temperature Sensor	3	3	Sense outdoor temperature
Air Conditioner	4	11	Turn on (with different modes) or turn off the air conditioner

consumption of the air conditioner influenced by the uncertain factors including ambient temperature, human activities and specific events. To enable the performance evaluation and comparison, we extended the original ThingML design to incorporate corresponding variation-related information.

The air conditioner plays a central role in the heating control system. It has four locations (i.e., *start*, *off*, $full_{on}$ and $half_{on}$) and 11 transitions. The *start* is an initial location which is marked with urgent. It is used to initialize environment parameters. The *off* location indicates that the current power is 0. The air conditioner can stay in the *off* location only when one of the following conditions holds: i) there is no person in the room; ii) the indoor temperature (monitored by the indoor temperature sensor) is higher than $12°C$; or iii) the alarm event is taking effect. To save energy, the air conditioner has two locations (i.e., $full_{on}$ and $half_{on}$) which provides different levels of power consumption. If the difference between indoor temperature and outdoor temperature is smaller than $2°C$, the air conditioner will work at a higher power level in the location $full_{on}$. If the difference between indoor temperature and outdoor temperature is equal to or smaller than $2°C$ and none of the *off* conditions holds, the air conditioner will work at a lower power level in the location $half_{on}$. To model the uncertain ambient temperature monitored by the outdoor temperature sensor, we use the formula $(-1) \times 6 \times cos(\pi \cdot globalClock/720) + base$. The value of the *base* indicates the initial room temperature at the beginning of a day. It follows the uniform distribution within range $[0°C, 7°C]$. Besides the outdoor temperature, the human activities strongly affect the energy consumption. To model uncertain human activities, we use the person detector to periodically check whether a person is in the office during the office hours (i.e., from 8:00 AM to 5:00 PM). In this experiment, we assume that the presence probability of a person is 70% during the office hours, while the presence probability of person near the office hours (i.e., from 7:00 AM to 8:00 PM, or from 5:00 PM to 6:00 PM) is 20%. Otherwise, the presence probability of a person in the office is 0. During the office time, for each hour there is a 70% chance that the window can be opened automatically to air the room, and the window will be kept open for ten minutes. If the alert component detects that the window is open, it will shut down the air conditioner to save energy.

In the ThingML design, all the sensors (i.e., indoor temperature sensor, outdoor temperature sensor, alarm component) are triggered by a timer periodically. Note that different timer-based interruptions will lead to different sensor sampling rates, which in turn affect the overall performance of the heating control system. In this example, we restricted the evaluation time of the heating control system to 10 days (i.e., 14400 minutes). To compare the performance of different timers for the heating control system under uncertain environments, we adopted a cost query in the form of $Pr[<= 14400](<> energy > 15000)$ to check *" what is the probability that the heating control system consumes more than 15000 units of energy within 10 days?"*

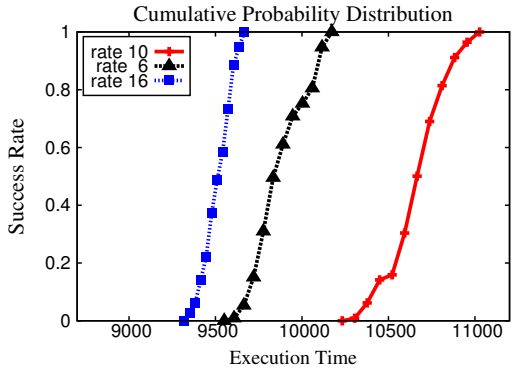

Figure 8. Quantitative analysis results with different sampling rates

Based on the same design, we conducted the experiments using three timers with different sampling times (i.e., 6, 10, and 16 minutes). Figure 8 compares the performance of the three design alternatives. By simulating 166 stochastic runs, UPPAAL-SMC can achieve a success ratio within range [0.95,0.99] with a confidence of 98%. The UPPAAL-SMC based evaluation takes 1756 seconds to get the quantitative analysis results. From this figure, we can observe that the design with a sampling time of 10 minutes achieves the best performance, since under the same energy constraint (i.e., 15000 energy units) the design with the 10-minute sampling rate has longer lasting time than the other two alternatives. Interestingly, we can note that the best sampling time here is neither the smallest one (i.e., 6 minutes) nor the largest one (i.e., 16 minutes). This is because if the sampling rate is higher, the system will become more sensitive to the uncertain environments. The frequent control state switches

may result in a waste of energy. Conversely, if the sampling rate is lower, the system reaction to environmental changes will become slower. As a result, overheating of the air conditioner will not be detected in time and therefore more energy will be consumed.

VI. CONCLUSIONS

As a promising modeling language, ThingML can significantly facilitate the model-driven development of IoT applications. However, since the current version of ThingML does not consider the impact of the surrounding uncertain environment, it is hard for designers to guarantee the QoS of IoT applications generated from ThingML design models. To address this problem, this paper presents a comprehensive system-level approach that enables the variation-aware modeling and quantitative analysis of ThingML designs. It introduces a set of newly defined modeling constructs for ThingML to enable the description of variation information. By using our proposed transformation rules, extended ThingML designs can be automatically transformed into NPTA models to conduct the quantitative evaluation against specified performance queries. Experimental results using two case studies demonstrate that our approach can effectively conduct the QoS evaluation and comparison for variation-based ThingML designs, which significantly facilitate the decision making of IoT designers.

ACKNOWLEDGMENT

This work was supported by the grants from Natural Science Foundation of China (Nos. 91418203 and 61402178), Innovation Program of Shanghai Municipal Education Commission 14ZZ047, and STCS Grants 14YF1404300.

REFERENCES

[1] L. Atzori, A. Iera and G. Morabito. The internet of things: A Survey. *International Journal of Information Systems Frontiers (ISF)*, 54(15):2787-2805, 2010.

[2] J. Gubbi, R. Buyya, S. Marusic and M. Palaniswami. Internet of things: A Vision, Architectural Elements, and Future Directions. *International Journal of Grid Computing-Theory Methods and Applications*, 29(7):1645-1660, 2013.

[3] F. Fleurey, B. Morin, A. Solberg and O. Barais. MDE to Manage Communications with and between Resource-Constrained Systems. *International Conference on Model Driven Engineering Languages and Systems (MODELS)*, 349-363, 2011.

[4] M. Kwiatkowska, G. Norman and D. Parker. PRISM 4.0: Verification of Probabilistic Real-time Systems. *International Conference on Computer Aided Verification (CAV)*, 585-591, 2011.

[5] M. Kwiatkowska, G. Norman and D. Parker. Stochastic Model Checking. Formal Methods for the Design of Computer, Communication and Software Systems: Performance Evaluation (SFM), 220-270, 2007.

[6] E. M. Clarke, O. Grumberg and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.

[7] A. David, K. G. Larsen, A. Legay, M. Mikucionis D. B. Poulsen, J. Van Vliet, and Z. Wang. Statistical Model Checking for Networks of Priced Timed Automata. *International Conference on Formal Modeling and Analysis of Timed Systems (FORMATS)*, 80-96, 2011.

[8] A. David, K. G. Larsen, A. Legay, M. Mikucionis and D. B. Poulsen. Uppaal SMC Tutorial. *International Journal on Software Tools for Technology Transfer (STTT)*, 17(4): 397-415, 2015.

[9] S. Huang, M. Chen, X. Liu, D. Du and X. Chen. Variation-Aware Resource Allocation Evaluation for Cloud Workflows using Statistical Model Checking. *International Conference on Big Data and Cloud Computing (BDCloud)*, 201-208, 2014.

[10] M. Chen, D. Yue, X. Qin, X. Fu and P. Mishra. Variation-Aware Evaluation of MPSOC Task Allocation and Scheduling Strategies using Statistical Model Checking. *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 199-204, 2015.

[11] A. David, K. G. Larsen, A. Legay, M. Mikucionis and Z. Wang. Time for Statistical Model Checking of Real-Time Systems. *International Conference on Computer-Aided Verification (CAV)*, 349-355, 2011.

[12] K. G. Larsen. Priced Timed Automata and Statistical Model Checking. *Int. Conf. on Integrated Formal Methods (IFM)*, 154-161, 2013.

[13] L.D. Xu, W. He and S. Li Internet of Things in Industries: A Survey. *IEEE Trans. on Industrial Informatics (TII)*, 10(4):2233-2243, 2014.

[14] S. Sendall and W. Kozaczynski. Model Transformation the Heart and Soul of Model-Driven Software Development. *IEEE Software*, 20(5):42-45, 2003.

[15] Internet of Things - Architecture (IoT-A). http://www.iot-a.eu/

[16] H. Yang and S. Pan. Modeling and Analysis of IOT Real-Time System using TCPN. *Information Technology Journal*, 12(9):1707-1716, 2013.

[17] ThingML. http://www.thingml.org/

[18] A. Legay, B. Delahaye and S. Bensalem. Statistical Model Checking: An Overview. *International Conference on Runtime Verification (RV)*, 122-135, 2010.

[19] D. Du, M. Chen, X. Liu and Y. Yang. A Novel Quantitative Evaluation Approach for Software Project Schedules using Statistical Model Checking. *International Conference on Software Engineering (ICSE) Companion*, 476-479, 2014.

[20] PingPong. https://github.com/HEADS-project/training/tree/master/1.ThingML_Basics/2.PingPong

[21] Heating Control System. https://github.com/HEADS-project/training/tree/master/1.ThingML_Basics/6.CEP

[22] OMNeT++ 4.6. https://omnetpp.org/