Quantitative Timing Analysis of UML Activity Diagrams Using Statistical Model Checking

Fan Gu[‡], Xinqian Zhang[‡], Mingsong Chen[‡], Daniel Große[§], and Rolf Drechsler[§]
[‡]Shanghai Key Lab of Trustworthy Computing, East China Normal University, Shanghai, China
[§]Institute of Computer Science, University of Bremen & Cyber-Physical Systems, DFKI GmbH, Bremen, Germany
Email: {fgu,xqzhang,mschen}@sei.ecnu.edu.cn; {grosse,drechsler}@informatik.uni-bremen.de

***Abstract*—Unified Modeling Language (UML) activity diagrams are widely used in modeling the dynamic aspects of system designs. However, due to frequent interactions between systems and external uncertain environment, the current version of UML activity diagrams cannot be used to accurately capture and quantify the overall timing behaviors of complex systems. To address this issue, this paper extends the UML activity diagrams to enable the stochastic modeling of user inputs and action executions, which strongly affect the overall timing behaviors of systems. Based on the statistical model checker UPPAAL-SMC, this paper proposes an automated framework that can perform quantitative reasoning under various functional and non-functional queries. Experimental results demonstrate the effectiveness of our proposed approach.**

I. Introduction

Due to the increasing interactions with external uncertain physical environment, the design complexity of *cyber-physical systems* (CPS) [1] is skyrocketing. How to model system behaviors within uncertain environment and how to guarantee the critical functional, real-time and performance requirements of specifications are becoming major challenges in CPS design. As a kind of behavior specification based on the Petri-net like semantics, UML activity diagrams are widely used in describing the concurrent behaviors of systems [2]. To guarantee the correctness and performance of activity diagrams, various model checking-based approaches [3]–[5] are proposed. However, most of them focuses on safety problems which can only answer "yes" or "no" based on given properties. Few of them can model and reason on the stochastic behaviors of activity diagrams under user input and action execution variations. For example, for an activity diagram, designers would like to ask the question "*What is the probability that a specified scenario can be triggered within time x*?". However, due to the nondeterministic execution and accumulated time variations, it is hard to figure out the answer and explain how to improve the performance under time variation. Clearly, the bottleneck lies in the lack of stochastic semantics supported by activity diagrams as well as effective ways to support the quantitative analysis.

To address the above problems, this paper makes two major contributions: i) we extend the syntax and semantics of UML activity diagrams to support the stochastic modeling of user inputs as well as the execution time of actions; and ii) we propose an automated framework that supports the quantitative timing analysis of activity diagrams based on *Statistical Model Checking* (SMC) [6]. Relying on monitoring random simulation runs of activity diagrams and the analysis using statistical methods (i.e., sequential hypothesis testing or Monte Carlo simulation), SMC can be used to estimate the satisfaction probability of user-specified performance queries. Unlike exhaustive model checking, SMC requires far less checking time and memory. Therefore, it is very scalable in validation of a wide spectrum of quantitative performance properties. Due to the flexible syntax and semantics provided by the SMC checker UPPAAL-SMC [7], we use it as the engine of our framework.

This work was partially supported by National Natural Science Foundation of China (Nos. 91418203 and 61202103), Innovation Program of Shanghai Municipal Education Commission 14ZZ047, German Academic Exchange Service (DAAD) in the PPP 57138060, and German Research Foundation (DFG) within the Reinhart Koselleck project DR 287/23-1. Mingsong Chen is the corresponding author.

The rest of this paper is organized as follows. After the introduction of related works in Section II, Section III introduces the notations of priced timed automata and SMC. Section IV gives the details of our approach. Section V presents two case studies to demonstrate the efficacy of our approach. Finally, Section VI concludes the paper.

II. Related Work

Model checking techniques have been widely investigated to analyze activity diagrams. For example, Li et al. [4] analyzed the timing behaviors of activity diagrams by using linear programming together with integer time verification techniques. Eshuis [8] presented a NUSMV-based approach that allows the consistency checking between activity diagrams and corresponding class diagrams. Hilken et al. [9] proposed a novel verification methodology that can ensure the consistency between activity diagrams and their contracts. In [10], Das et al. proposed an activity diagram-based method that can verify real-time MPSoC applications. Although existing model checking-based approaches can be used to enhance the reliability of systems, most of them focus on the analysis of safety properties. Few of them support the quantitative analysis of activity diagrams.

As a promising approach, statistical model checking [6], [7] has been applied to evaluate variation-aware designs. For example, Du et al. [13] utilized the UPPAAL-SMC tool to evaluate project schedules with time uncertainty. Chen et al. [12] presented a method that can evaluate the task allocation and scheduling strategies with time and power variation information. However, the above approaches only consider the scheduling of tasks organized in the DAG form without loops, which is much more simpler than activity diagrams. To the best of our knowledge, our proposed approach is the first attempt that uses SMC for the analysis of activity diagrams considering both input and execution time variations.

III. Background of NPTA and SMC

Our approach adopts the *Network of Priced Timed Automata* (NPTA) [7] to model the stochastic behaviors of activity diagrams. PTAs are a variant of timed automata whose clocks can evolve with different rates in different locations. An NPTA consists of a set of correlated PTAs that communicate with each other using broadcast channels and shared variables. For example, Figure 1 shows an NTPA denoted by (A|B) with two PTAs A (*id=ida*) and B (*id=idb*), where each PTA has four locations and two clocks (e.g., C_a and $c1$ for A). In different locations, the values of primed clocks indicate the rates of different continuous variables. For example, $C_a' == 2$ in location A_1 indicates that the rate of C_a is 2 in A_1. By default, unprimed clocks have a rate of 1. To enable the message-based synchronization between PTAs, we use a channel array *msg[id]*, where *id* indicates the PTA identifier of the message target. While using broadcasting-based synchronization, we adopt the *non-deterministic selections* to

filter useless messages. For example in PTA *B*, the *selections e:msg_t* and guard condition *e==idb* are used to monitor incoming messages and filter messages which are not sent to B.

Although UPPAAL-SMC only supports the uniform and exponential distributions explicitly, by proper usage of the built-in function *random()*, we can produce values that follow a large set of commonly used distributions. For example, based on the *Box-Muller* method, we can generate the normally distributed random values using the *random()* function. To model the stochastic behaviors of user input as well as the execution time of each action, we adopt the pattern as shown in Figure 1, where *T_Dist()* and *V_Dist()* are used to generate time delays and variable values following some specific distributions. Since location A_1 sets an upper bound for clock c1 (i.e., $c1 <= t1$) and its outgoing transitions set a guard condition $c1 >= t1$, PTA *A* can stay in location A_1 with a delay of *t1* which is randomly generated by *T_Dist(ida)*. Meanwhile, due to the random value of *v*1 generated by *V_Dist()*, the stochastic behaviors of (A|B) will be strongly affected by these two factors. It is important to note that we did not extend the semantics of NPTA here. Based on the above pattern, arbitrarily complex input and time variation aware stochastic behaviors can be modeled.

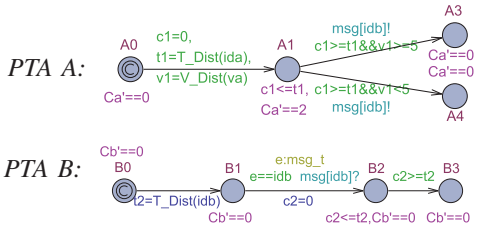


Fig. 1. An NPTA, (A|B)

During the checking of NPTA models, SMC simulates random runs which are bounded by either time, cost or a number of discrete steps. Upon a decision of an NPTA during the simulation, the transition with the shortest delay will be triggered and all the continuous variables will be updated. Note that commit and urgent locations (i.e., locations marked with the symbol C or U) within a PTA have delay of 0 and the outgoing transitions of commit locations have higher priority. All the derived runs are monitored by specified properties in the form of cost-constraint temporal logic formulas [7]. Our approach adopts the properties in the form of $Pr[time <= bound](<> expr)$, where *bound* is a constant value which denotes the time limit, and the expression $<> expr$ asserts that eventually the state predicate *expr* will happen. By replacing the time limit and *expr* with specified value and desired functional scenarios, we can conduct different kinds of performance queries for the timing analysis of activity diagrams as described in Section IV-C. Finally, the checker will report the interval estimate for the success ratio of the given property.

IV. Our Approach

Figure 2 shows the overview of our framework and the workflow of our approach. Initially, activity diagrams coupled with our extended variation information (e.g., action execution time variation, input value variation) are parsed by our developed tool to generate the corresponding NPTA models. In our approach, NPTA models are divided into two parts: i) front-end models which describe the common behaviors of nodes (action nodes and control nodes) in activity diagrams; and ii) back-end models which specify the different features (i.e., flow relation, variation information, operation definitions) of activity diagrams. To allow for the quantitative analysis of activity diagrams via performance queries, we translate design requirements into properties in the form of cost-constrained temporal logic [6]. When both the NPTA models and performance query-based properties are ready, our framework employs the checker UPPAAL-SMC to conduct the quantitative analysis of stochastic system behaviors. In the following subsection, we will explain the major components of our framework in detail.

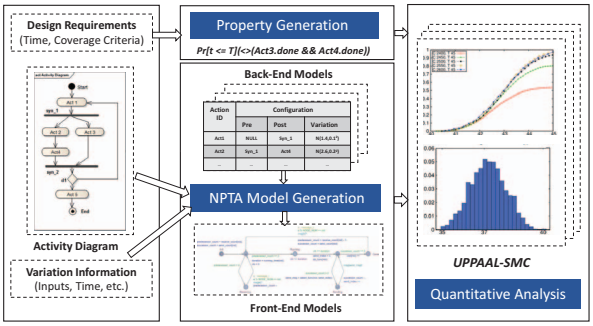


Fig. 2. An overview of our framework

A. Semantics Extension for Activity Diagrams

Our approach uses UML 2.x as our specification, where activity diagrams adopt the Petri-net like semantics [2]. Figure 3 shows an example of an activity diagram. The diagram describes a scenario of withdrawing money from an ATM. Initially, users are required to input their access code. They only have two chances. If both access code inputs fail, the withdraw process will be terminated. Next, ATM will ask users to input the amount of the money he/she wants to withdraw. Meanwhile, the printer will be warmed. Based on the deposit, the ATM decides whether to dispense the cash. Finally, the receipt of the transaction will be printed.

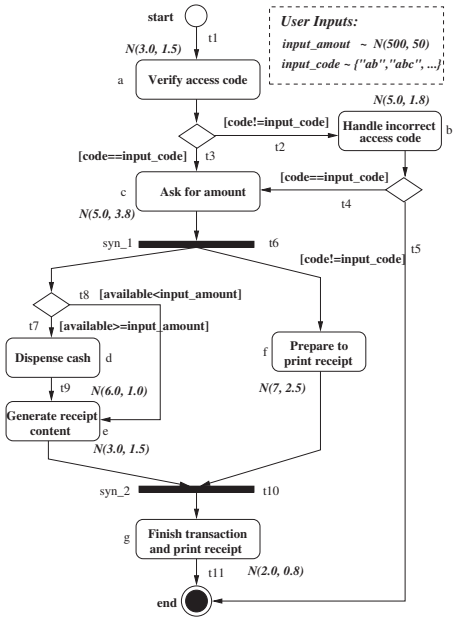


Fig. 3. The UML activity diagram of an ATM

The illustrative example in Figure 3 consists of majority of basic constructs of activity diagrams. Based on the semantics of Petri-net, actions denoted by round cornered boxes represent the execution of operations on input tokens, and newly generated tokens will be delivered to corresponding outgoing edges. For example, action *a* denotes the operation *verify access code*, which can be abstracted as a procedure that deals with the checking of input access code. In activity diagrams, there are two kinds of flows: i) control flow which indicates the execution sequence of actions, and ii) object flow which denotes the relation of data token transmission. To simplify the description of activity diagrams, we combine both control and data tokens together. Therefore, we do not distinguish control flow

edges and object flow edges in our framework. A flow of an activity diagram starts from the initial node (e.g., the node *start*) where tokens are constructed and initiated, and the flow ends at a final node (e.g., the node *end*) where all tokens are destroyed. Decision/merge nodes use the same diamond-shaped symbol. While decision nodes choose one of the outgoing flows with satisfying guard predicates, merge nodes select and deliver one of incoming flows to a next activity node. To describe concurrent behaviors, fork nodes (e.g., *syn_1*) and join nodes e.g., (*syn_2*) are used with multiple arrows leaving or entering synchronization bars, respectively. When a token reaches a fork node, it will be duplicated and forwarded to all consequent outgoing actions. Join nodes synchronize multiple flows by merging multiple tokens into one token. The join node can be activated only when the tokens of all incoming edges are available.

To enable stochastic behavior modeling of concurrent systems, we extend the syntax and semantics of activity diagrams. The current version of our approach supports the stochastic modeling of user inputs and timing of actions in activity diagrams. It allows different kinds of distributions for both user inputs and action execution time modeling. As shown in Figure 3, the user input *input_amount* follows a normal distribution, while *input_code* follows a uniform distribution on an enumerated type. For each action, we can assign it with a time distribution to indicate the statistics of operation execution time. For example, the execution time of action a follows the normal distribution $N(3.0, 1.5)$, which indicates that its mean execution time is 3.0 seconds and the standard deviation is 1.5 seconds.

Since activity diagram itself is a semi-formal specification which cannot be directly used for model checking, we use an extended Petri-net as an intermediate formal model to capture the stochastic behaviors of activity diagrams as well as guide the translation to UPPAAL-SMC inputs. Definition 1 gives the formal definition of the structural information of our stochastic activity diagrams. In the formalization, we use the *completion transition* and *flow edge* to model the concurrent behaviors of systems. If a completion transition has multiple incoming flow edges, it will do the join operation. If a completion transition has multiple outgoing flow edges, then it will do the fork operation.

Definition 1. *A stochastic activity diagram is a tuple AD=(A, M_{act}, T, F, C, V, VAL, V_{ipnut}, M_{val}, M, E, a_I, a_F) where*

- $A = \{a_1, a_2, \ldots, a_m\}$ *is a set of action nodes.* $M_{act} : A \to DIST_t$ *is a mapping that specifies the distributions of the execution time of actions.*
- $T = \{t_1, t_2, \ldots, t_n\}$ *is a set of completion transitions.*
- $F \subseteq \{A \times T\} \cup \{T \times A\}$ *is the set of flow edges.*
- $C = \{c_1, c_2, \ldots, c_n\}$ *is a finite set of guard conditions.* $Cond : F \to C$ *is a mapping that assigns each flow edge* $f_i \in F$ *with a guard condition* c_i.
- *Let* $V = \{V_1, V_2, \ldots, V_k\}$ *be the set of all variables used by AD, VAL be the set of all possible variable assignments, and* $V_{input} \subset V$ *be the input variables whose values rely on user inputs.* $M_{val} : V_{input} \to DIST_v$ *is a mapping that specifies the value distributions of input variables.*
- $M : A \times 2^{V \times VAL} \to 2^{V \times VAL}$ *is a mapping that describes the value changes of variables within an action.*
- $a_I \in A$ *is the initial node, and* $a_F \in A$ *is the final node. There is only one completion transition* $t \in T$ *s.t.* $(a_I, t) \in F$, *and for any* $t' \in T$, $(t', a_I) \notin F$ *and* $(a_F, t') \notin F$.

To analyze the dynamic behaviors of an activity diagram, we use the concurrent states to indicate the simultaneously executing actions and their execution time.

Definition 2. *Let AD be a stochastic activity diagram. The current state (CS) of AD is a pair* (ν, θ), *where* $\nu \subseteq A$ *and* $\theta : \nu \to \mathbb{R}_0^+$ *is a clock function that indicates how long the action* $a \in \nu$ *has started. The initial state of AD is* (a_I, θ_0) *where* $\theta_0(a_I) = 0$, *and any state* (a_F, θ') *is a final state of AD.*

Definition 3. *Let* $t \in T$ *be a transition and* (ν, θ) *be a state of an activity diagram.*

- ${}^\bullet t$ *denotes the preset of t, then* ${}^\bullet t = \{a \mid (a,t) \in F\}$.
- $t^\bullet$ *denotes the postset of t, then* $t^\bullet = \{a \mid (t,a) \in F\}$.
- $enabled(\nu)$ *denotes the set of completion transitions that are associated with the outgoing flow edges of* ν, *then* $enabled(\nu) = \{ t \mid {}^\bullet t \subseteq \nu \}$.
- $firable(\nu)$ *denotes the set of transitions that can be fired from CS , i.e.,* $firable(\nu) = \{ t \mid t \in enabled(\nu) \bigwedge {}^\bullet t$ *are all completed* $\bigwedge \exists\, n \in A.\ Cond((t,n))$ *is satisfied* $\bigwedge (\nu - {}^\bullet t) \cap t^\bullet = \emptyset \}$.

During the concurrent execution of actions, when an action finishes, it needs to determine the next state. Note that at this time, although it is required that action executions should follow the specified time distributions, when none of the completion transitions can be fired, the change of ν in the current state will be delayed. For example, upon the completion of node f in Figure 3, the join node *syn_2* may not be executed immediately, since the join node needs to wait for the completion of all the incoming nodes.

Definition 4. *Let AD be a stochastic activity diagram and* (ν, θ) *be the current state. If upon the completion of the action* $\alpha \in \nu$ *there exists a transition* $\tau \in T$ *such that* $(\alpha, \tau) \in F$ *and* $\tau \in firable(\nu)$, τ *will be fired immediately and the value of* $\theta(\alpha)$ *should strictly follow the distribution of* $M_{act}(\alpha)$. *Otherwise,* α *needs to wait until* τ *is firable. Let* δ *be the delay since the last fire of some transition. The new state* $(\nu', \theta') = fire((\nu, \theta), \tau, \delta)$ *can be derived by:*

1) $\nu' = (\nu - {}^\bullet\tau) \bigcup \tau^\bullet$, *and*
2) $\forall\ \alpha \in \nu',\ \theta'(\alpha) = \begin{cases} 0 & \alpha \in \nu' - (\nu - {}^\bullet\tau) \\ \theta(\alpha) + \delta & otherwise \end{cases}$.

The behavior of stochastic activity diagrams can be represented by a sequence of states and fired transitions together with delays.

Definition 5. *A run* ρ *of a stochastic activity diagram AD is a sequence of states, transitions and delays, i.e.,*

$$\rho = (\nu_0, \theta_0) \xrightarrow{(\tau_0, \delta_0)} (\nu_1, \theta_1) \xrightarrow{(\tau_1, \delta_1)} \ldots \xrightarrow{(\tau_{n-1}, \delta_{n-1})} (\nu_n, \theta_n)$$

where $\nu_0 = \{a_I\}$, $\theta_0(a_I) = 0$, $\nu_n = \{a_F\}$, $\theta_n(a_F) \in \mathbb{R}_0^+$ *and* $(\nu_{i+1}, \theta_{i+1}) = fire((\nu_i, \theta_i), \tau_i, \delta_i)$. *In the run,* δ_i *denote the time interval between the* i*th and* $(i+1)$*th completed actions.*

B. NPTA Model Generation

We extend the semantics of activity diagrams to incorporate both the distribution information and the action implementation details. Since the original activity diagrams do not support these features, in our approach all such information is saved as a UML note for activity diagrams. Based on the formal definitions in Section IV-A, activity diagrams can be parsed and automatically translated into an executable NPTA model for the purpose of quantitative analysis. Similar to the work proposed in [12], our approach decouples the syntax and semantics of activity diagrams using the front-end model and back-end configuration. Note that all the activity diagrams share the same front-end model but different back-end configurations which specify the parameters as well as data structures to support the stochastic execution of activity diagrams.

1) Back-end Model: During the translation, we abstract an activity digram as a directed graph with action nodes and control nodes (i.e., decision and merge nodes) connected by flow edges. To simplify the modeling, in the directed graph we do not model the synchronization

bars explicitly. Instead, we put the constraint that a node can be executed only when all its precedent nodes are complete. Such synchronization constraint is implemented based on the communication between nodes. When one node is complete, it will notify all its successive nodes. When an action node collects all the notifications from its predecessor nodes, it can start to execute.

Unlike front-end model which has a graphical representation, the back-end configuration is textual. It defines global constants and variables (distribution information and user inputs), structure information (node precedence and message channels) and functions (action functions and branch functions). By using our developed parsing tool, such information can be automatically extracted from the extended activity diagrams and transformed into back-end configurations.

Assume that there are N nodes (i.e., action nodes and control nodes) in an activity diagram. To identify each node, the back-end configuration assigns each node with an ID. To describe the execution time distributions of these nodes, a two-dimensional array *distribution[N][m]* is used. Assume that the ID of current node is *nid* and the execution time follows the normal distribution. The expected execution time of current node is saved in *distribution[nid][0]*, and its standard deviation is saved in *distribution[nid][1]*. Since control nodes only handle the control flow, we assume that they have a delay of 0. To allow the modeling of action loops in a run, we use an array *visit[N]* to record the visit number of each node.

In the extracted directed graph, the directed edges indicate the token flow. In other words, they reflect the notification message flow. For example, if there is a flow edge from node id_x to node id_y, it means that, after the execution of node id_x, node id_x will immediately send a notification to node id_y. Since UPPAAL-SMC only supports the broadcast for the communication, to create a private channel to support the point-to-point communication, we create an urgent channel array *msg[N × N]*. We encode the communication from node id_x to node id_y using the formula $encode\_msg(id_x, id_y) = id_x \times N + id_y$, where $msg[encode\_msg(id_x, id_y)]$ indicates the private unidirectional channel from node id_x to node id_y. To model all the flow edges in the back-end configuration, we do not construct a matrix of size $N \times N$. Similar to the adjacent list, we use a two-dimensional array *msg_graph[N][MAX_OUT]* to save all the flow edge information, where *MAX_OUT* indicates the maximum number of output flow edges of all the nodes. Note that *msg_graph[i][j]* indicates the j_{th} message channel that starts from node *i* rather than the message channel from node *i* to node *j* If *msg_graph[i][j]==-1*, it means that there are at most *j* flow edges that start from node *i*. Otherwise, it means that there is a message sent from node *i* to node $msg\_graph[i][j]\%N$. Since different nodes have different number of precedent nodes and successive nodes, we use the *receive_count[N]* and *send_count[N]* to save such information respectively. Before execution, the current node should collect *receive_count[nid]* notifications. Once the current node is complete, it will notify *send_count[nid]* successive nodes using the information saved in between *msg_graph[nid][0]* and *msg_graph[nid][send_count[nid]-1]*.

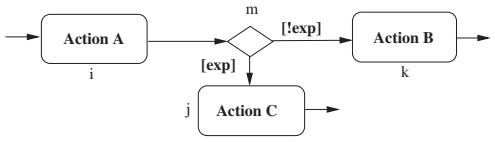


Fig. 4. A snippet of an activity diagram

In the translation, all variables of activity diagrams are made global. Actions can be considered as functions that deal with these variables, and control nodes can be considered as branch functions that determine the flow of messages (i.e., tokens). For input variables, the back-end configurations defines its value distribution, and their random values are generated in the initial action (i.e., a_I). Since UPPAAL-SMC supports almost the same programming constructs as in C programming language, the translation of an action function to its counterpart in UPPAAL-SMC needs few modifications. For each action with an ID *nid*, in the back-end configuration, we will create one *action function* counterpart named *act_func_nid()*. For example in Figure 4, the action *A* has a corresponding UPPAAL-SMC action function *act_func_i()*. To facilitate the usage of UPPAAL-SMC functions in the front-end model, we define the function *do_func(ID)* in the back-end configuration which can call any action function with its ID. If *nid* refers to a control node, *do_func(nid)* will return immediately without doing anything. For each control node (i.e., decision or merge node), we create a *branch function br_func_nid()* in the back-end configuration. For example in Figure 4, the decision node *m* has a branch function *br_func_m()*. Since the node can only choose one output, we can get the information such that *send_count[m]=1*, *receive_count[m]=1*, *msg_graph[m][0]=m × n + k*, *msg_graph[m][1]=m × n + j* and *msg_graph[m][x]=-1* for $x > 1$. The major purpose of branch function is to determine which flow edge will be chosen. Listing 1 gives an overview of the definition of the branch function of decision node *m* in Figure 4. It also presents a *unified branch function select_func()* which can call any branch functions based on the ID of branch nodes.

```
message_t br_func_m(id_t nid){
    if(exp)  return msg_graph[nid][0];
    if(!exp) return msg_graph[nid][1];
    else  return -1;
}
message_t br_func_n(id_t nid);
......
message_t select_func(id_t nid){
    if(nid==m) return br_func_m(nid);
    if(nid==n) return br_func_n(nid);
    ......
    return -1;
}
```

Listing 1. An overview of branch functions in back-end configuration

2) Front-end Model: Without considering the internal operation and timing information, the behaviors of action and control nodes can be modeled uniformly. After all precedent nodes have finished, the current node can start. When a node is complete, it will notify all its successive nodes immediately. Assume that the current action or control node has an ID *nid*. Figure 5 shows the front-end model for a single action or control node. The model shows that each node has five major states: i) The *init* state is a commit state that initializes the count information of the precedent and successive nodes. ii) After initialization, the *receiving* state tries to collect all the notifications (i.e., tokens) from all its precedent nodes via the channels in *msg[]*. iii) Upon receiving all the notifications from precedent nodes, the *running* state executes the current action implemented in function *do_func* with a time delay *duration* which is randomly generated following the given distribution. iv) Once the current node is complete, the *sending* state will judge the guard expressions on the outgoing edges of current action or control node using the function *select_func* and notify the successive nodes accordingly. v) The *done* state increases the counter *visit[nid]* to indicate the finish of the current node. It is important to note that the node can be reactivated when receiving a notification by other nodes. Therefore, our framework allows the quantitative analysis of loops in the design.

C. Property Generation and Quantitative Analysis

Although UPPAAL-SMC is based on simulation, the statistical model checking borrows the idea of traditional model checking approaches. When NPTA models are constructed, the designers should provide properties for the purpose of analysis. Since our

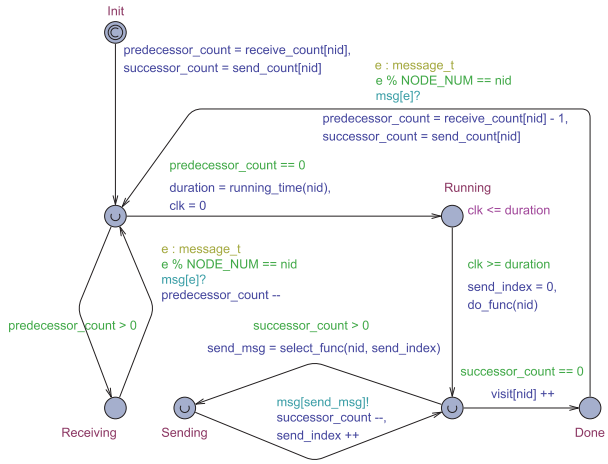


Fig. 5. Front-end model for activity diagrams

approach focuses on the quantitative analysis of activity diagrams, the designers would like to query "*what is the probability that a functional scenario can happen or complete within a time limit?*" Assume that the function scenario can be described using the formula ψ, UPPAAL-SMC supports the probability estimation for properties in the form of $Pr[<=bound](\psi)$, where the *bound* specifies the time limit. For example, $Pr[<=T](<>act.done)$ tries to figure out the probability that the action *act* can complete within time T.

Our approach focuses on the time analysis of the stochastic behavior of activity diagrams. Note that our framework can be easily extended to deal with other kinds of performance analysis. Assuming that $p = Pr[<=bound](<>\psi)$, based on the parameters ε (probability uncertainty) and α (probability of false negatives) provided by designers, UPPAAL-SMC [7] computes the number of runs needed in order to produce an approximation interval $[p-\varepsilon, p+\varepsilon]$ with a confidence $1-\alpha$. When the simulation-based check finishes, the distribution of the probability of successful simulations will be reported to enable the quantitative analysis.

Coverage oriented analysis are widely used in the analysis of activity diagrams. Inspired by the coverage metrics (i.e., action coverage, interaction coverage, and key path coverage) presented in [5], our framework can automatically extract such functional coverage information from activity diagrams and translate it into performance queries (i.e., properties) accordingly. By default, our framework supports the following three kinds of performance queries, which can fulfill most common purposes.

- **Action queries** are in the form of $Pr[<=T](<> act_i.status \;\&\&\; visit[i] >= k)$, which evaluates the possibility that within time limit T the action act_i can be visited with the state *status* (e.g., done, running, receiving) at least k times.
- **Interaction queries** are in the form of $Pr[<=T](<> act_i.status \;\&\&\; act_j.status)$, which evaluates the possibility of concurrent actions with specified statuses within time limit T.
- **Run queries** are in the form of $Pr[<=T](<> act_{\tau_0}.done \;\&\&\; act_{\tau_1}.done \;\&\&\; \ldots \;\&\&\; act_{\tau_{n-1}}.done \;\&\&\; visit[v_0] >= k_0 \;\&\&\; \ldots \;\&\&\; visit[v_{n-1}] >= k_{n-1})$ where act_{τ_i} is the action (with an ID of v_i) that has the outgoing flow edge τ_i. The queries try to figure out the probability that the run can be finished within time limit T.

While there is no specific constraint on loop times, the *visit* value is set to 1 by default for the above queries. It is important to note these three kinds of queries are by no means the golden ones for the quantitative analysis. Other kind of queries are allowed in our framework for specific purposes.

V. Case Study

Based on our proposed framework, we developed a tool chain that integrates the UML edit tool Enterprise Architect, UPPAAL-SMC model checker and our NPTA model generator (implemented using the JAVA programming language). Based on the given user requirement, the generator can parse extended activity diagrams and translate them into NPTA models and properties automatically.

To evaluate our approach, this section presents two case studies: an *Automatic Train Operation* (ATO) subsystem of the railway signaling system *Communication-Based Train Control* (CBTC) [11], and an *Online Stock Exchange System* (OSES) [5]. Although the original activity diagrams of both designs do not have the stochastic information, we include them based on the suggested data from our industrial parter. In the experiments, all the properties are generated automatically based on the coverage information specified by designers. Due to page limit, for the analysis results of each kind of queries, we only show the typical cases. The experimental results were obtained from UPPAAL-SMC V4.1.18 on a desktop with 3.30GHz AMD CPU and 4GB RAM.

A. Experiment 1 – Onboard Subsystem of CBTC

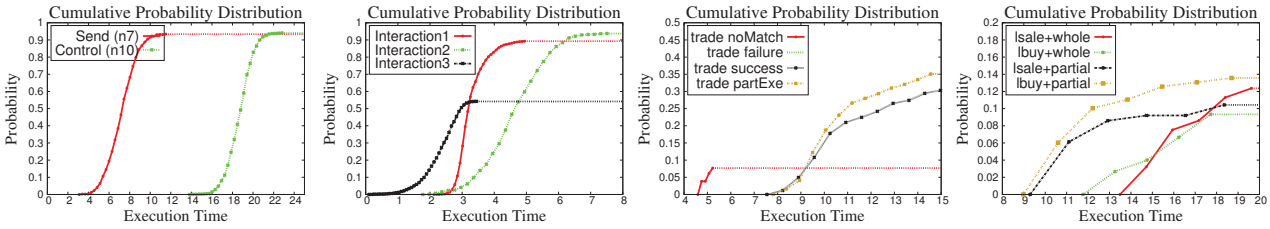
CBTC is a railway signaling system that makes use of the telecommunications between trains and track equipments for traffic management and infrastructure control. The ATO subsystem is an operational safety enhancement system which is used to help automate operations of trains. Since trains run within uncertain environments, the ATO subsystem suffers from the delay of communication and the execution time variations of software and hardware components.

TABLE I
Execution Time Distributions of ATO Actions

ID	Action Function	Time Distribution
n1	receive wireless communication signals	N(3.0, 0.2)
n2	calculate static speed curve	N(2.4, 0.4)
n3	select strict static speed curve	N(4.0, 0.9)
n4	calculate dynamic speed curve	N(1.5, 0.1)
n5	calculate train position	N(2.8, 0.8)
n6	generate train position report	N(1.8, 0.5)
n7	send signals	N(2.6, 1.0)
n8	compare with actual train position	N(3.6, 0.6)
n9	generate train control information	N(2.2, 0.2)
n10	control the train	N(2.0, 0.1)

We collect the activity diagram based design from [11], where the diagram has 10 action nodes (without counting the initial and final actions), 2 fork nodes and 2 join nodes. Since the response time of operations (actions) is considered as the most important issue in the system-level design, in this example we focus on the quantitative analysis of the execution time. We do not investigate the stochastic user inputs in this example. Table I presents the time distributions for each action. In this example, we assume each action execution time follows the normal distribution. The first column presents the ID of actions, and the second column present the function conducted in the action. The last column presents the execution time distribution of actions. For example, the execution time of action $n1$ follows the distribution $N(3.0, 0.2)$, where the expected execution time is 3.0 milliseconds and the standard deviation is 0.2 millisecond.

By applying the action queries, we can figure out the probability of an action within a time limit. For this experiment, we set both ε and α to 0.02 for UPPAAL-SMC. Figure 6(a) presents two action query instances (i.e., $Pr[<=25](<> n7.done)$ and $Pr[<=25](<> n10.done)$), which evaluate whether the action $n7$ and $n10$ can complete within 25 milliseconds. By running 890 runs, the first property obtains a probability interval $[0.91, 0.95]$ with a confidence 98%. By running 808 runs, the second property gets a probability interval $[0.92, 0.96]$ with a confidence 98%. Both quantitative analysis



(a) Action queries for ATO (b) Interaction queries for ATO (c) Action queries for OSES (d) Run queries for OSES

Fig. 6. Query results of ATO design and OSES design

cost around 5 minutes to get the evaluation results. From this figure, we can find that action $n7$ has a shorter response time than action $n10$. Interesting, we can find that in both cases when increasing the time limit after a threshold (e.g., 22 milliseconds for action $n10$), due to the accumulated time variations of actions, the change of the completion probability is quite small.

The ATO subsystem has quite a lot of concurrent executing components. To check the correlation between them, we adopt the interaction queries for the quantitative analysis. Figure 6(b) presents three typical interaction scenarios using the properties:

1) Scenario1: $Pr[<= 5](n2.running \;\&\&\; n6.running)$ which checks the overlapped execution between actions $n2$ and $n6$ within 5 milliseconds.
2) Scenario2: $Pr[<= 8](n7.running \;\&\&\; n4.receving)$ which checks the probability that action $n7$ happens before action $n4$ within 8 milliseconds.
3) Scenario3: $Pr[<= 5](n5.done \;\&\&\; n1.running)$ which checks the probability that action $n5$ completes before the completion of $n1$ within 5 milliseconds.

The evaluation time of all these three queries is less than 5 minutes. From the figures reported by UPPAAL-SMC, we can not only figure out the probability of each interaction scenario, but also analyze the change of the probability along the execution time.

B. Experiment 2 – Online Stock Exchange System

The activity diagram of the OSES design [5] is to model the stock transaction processing scenarios including accepting, checking, and executing the customers' orders (market orders and limit orders). It consists of 27 activities, 29 transitions and 18 key paths. In this example, we consider the distributions of both user inputs and action execution time. We assume that 50% of the orders are buy orders and 50% of the orders are sale orders. We also assume that 20% of the orders employ market price and 80% of the orders use the limit price. For a specific stock, the *market price*, *limit price*, *order amount* and *available amount* are all randomly generated using normal distributions. We also assume that the execution time of actions follows the normal distribution. Due to page limit, we do not present the distribution details.

In the OSES design, the quantitative timing analysis of action completion is a very important issue, since it can not only guarantee the proper user experience, but also can be used to detect the performance bottleneck of the system deployment. For this experiment, we set both ε and α to 0.05 for UPPAAL-SMC. Figure 6(c) shows the action queries results using four properties (in the form of $Pr[<= 15](<> act.done)$) to check whether the actions *trade_success*, *trade_failure*, *trade_noMatch* and *trade_partexe* can complete within 15 time units. Each query in this example costs around 2-hour SMC simulation time. From this figure we can find that the probability of the response of transaction failures within 15 time units is extremely low (no data generated). Among all simulation runs, the probability of activating the *noMatch* events is lower than 10%, and the *noMatch* action can abort the transaction much earlier (less than 6 time units) than the time limit. Under the stochastic input and execution time settings, the chance of partial execution is a little bit higher than the successful execution (35% versus 30%).

Since 80% of the orders are limit orders, our experiment focuses on the quantitative analysis of the limit trades. For the limit sale/buy orders in OSES design, there are two possibilities of the outcome: the order is fully traded or partially traded. To check the scenarios of four combinations, we conducted the quantitative analysis using the run queries. In these queries, we do not specify the activation number of actions. In other words, any executions with repeatable actions will be counted in the run analysis. Figure 6(d) shows the quantitative analysis results of the four combinations within a time limit of 20 time units. We can find that among all the transactions the case *lbuy+partial* can achieve the highest ratio. Interestingly, we can find that at time 20 the case *lsale+whole* has a higher chance to be completed than the case *lsale+partial*. However, if we set the time limit to be smaller than 18, we will obtain an opposite answer.

VI. Conclusions

Due to the increasing interaction between systems and surrounded environment, how to capture and analyze uncertain timing behaviors of systems is becoming a major bottleneck. To address this problem, this paper extended UML activity diagrams with the capability of stochastic modeling of timing behaviors. Furthermore, it proposed a framework based on UPPAAL-SMC that can conduct automated quantitative analysis of activity diagrams with a large spectrum of property-based performance queries. The experimental results of two industrial case studies demonstrate the efficacy of our approach.

References

[1] E. A. Lee and S. A. Seshia. Introduction to Embedded Systems - A Cyber-Physical Systems Approach. *LeeSeshia.org*, 2011.

[2] Y. Vanderperren, W. Mueller, and W. Dehaene. UML for Electronic System Design: A Comprehensive Overview. *Design Automation for Embedded Systems*, 12(4):261-292, 2008.

[3] R. Wille, M. Gogolla, M. Soeken, M. Kuhlmann and R. Drechsler. Towards a generic verification methodology for system models. *DATE*, 1193–1196, 2013.

[4] X. Li, C. Meng, Y. Pei, J. Zhao and G. Zheng. Timing Analysis of UML Activity Diagrams. *UML*, 62–75, 2001.

[5] M. Chen, et al. Efficient test case generation for validation of UML activity diagrams. *Design Automation for Embedded Systems*, 14(2):105-130, 2010.

[6] A. Legay and M. Viswanathan. Statistical model checking: challenges and perspectives. *STTT*, 17(4):369-376, 2015.

[7] A. David, K. G. Larsen, A. Legay, M. Mikucionis, D. Poulsen. Uppaal SMC tutorial. *STTT*, 17(4):397-415, 2015.

[8] Rik Eshuis. Symbolic model checking of UML activity diagrams. *ACM TOSEM*, 15(1):1-38, 2006.

[9] C. Hilken, J. Seiter, R. Wille, U. Kuhne and R. Drechsler:. Verifying consistency between activity diagrams and their corresponding OCL contracts. *FDL*, 1–7, 2014.

[10] D. Das, R. Kumar and P. P. Chakrabarti. Timing Verification of UML Activity Diagram Based Code Block Level Models for Real Time Multiprocessor System-on-Chip Applications. *APSEC*, 199–208, 2006.

[11] IEEE Rail Transit Vehicle Interface Standards Committee. IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements (1474.1-2004). 2004.

[12] M. Chen, D. Yue, X. Qin, X. Fu and P. Mishra. Variation-aware evaluation for MPSoC task allocation and scheduling strategies using statistical model checking. *DATE*, 199–204, 2015.

[13] D. Du, et al. A novel quantitative evaluation approach for software project schedules using statistical model checking. *ICSE Companion*, 476–479, 2014.