SCIENCE CHINA
Information Sciences

• **RESEARCH PAPER** •

March 2016, Vol. 59 032101:1–032101:17
doi: 10.1007/s11432-015-5332-8

Model based verification of dynamically evolvable service oriented systems

Yu ZHOU[1,2]*, Jidong GE[2], Pengcheng ZHANG[3] & Weigang WU[4]

[1]*College of Computer Science and Technology, Nanjing University of Aeronautics and Astronautics, Nanjing* 210016*, China;*
[2]*State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing* 210023*, China;*
[3]*College of Computer and Information, Hohai University, Nanjing* 210098*, China;*
[4]*College of Computer Science, Sun Yat-Sen University, Guangzhou* 510006*, China*

Received October 20, 2015; accepted December 2, 2015; published online January 21, 2016

Abstract Dynamic evolution is highly desirable for service oriented systems in open environments. For the evolution to be trusted, it is crucial to keep the process consistent with the specification. In this paper, we study two kinds of evolution scenarios and propose a novel verification approach based on hierarchical timed automata to model check the underlying consistency with the specification. It examines the procedures before, during and after the evolution process, respectively and can support the direct modeling of temporal aspects, as well as the hierarchical decomposition of software structures. Probabilities are introduced to model the uncertainty characterized in open environments and thus can support the verification of parameter-level evolution. We present a flattening algorithm to facilitate automated verification using the mainstream timed automata based model checker –UPPAAL (integrated with UPPAAL-SMC). We also provide a motivating example with performance evaluation that complements the discussion and demonstrates the feasibility of our approach.

Keywords dynamic evolution, verification, model checking, service oriented systems, timed automata

Citation Zhou Y, Ge J D, Zhang P C, et al. Model based verification of dynamically evolvable service oriented systems. Sci China Inf Sci, 2016, 59(3): 032101, doi: 10.1007/s11432-015-5332-8

1 Introduction

With the development of the network technology, the operating environment of modern software is not as isolated or closed as before. Such an environment, characterized by the openness and dynamism, inevitably affects the behaviors of the inhabitant systems, for example, the inherently uncertain nature of the network, the volatile user requirements as well as the quality of services. This ongoing trend has a significant implication on the software construction and calls for more flexible evolvability of software artifacts [1]. Service oriented systems are becoming a promising paradigm for software in the age of the Internet. Since the ultimate goal of software process is to deliver satisfying products without compromising the quality of service, the need for supporting dynamic evolution is becoming an essential concern, especially in some mission-critical domains [2].

Dynamic evolution denotes the process of changing behaviors according to the contextual information so as to provide continuous and desired quality of services. By behavior, we mean the collection of software

*Corresponding author (email: zhouyu@nuaa.edu.cn) execution sequences as well as the states and their transitions of software systems [3]. In [4], Wang et al. defined *dynamic evolution* as a specific kind of evolution that updates running programs without interrupting their execution. Thus the main purpose is to minimize the cost of service unavailability or to avoid service degradation. For the process to be trusted, consistency with specification is a prerequisite. As aforementioned, the software artifacts are usually composed of a set of autonomous services in open environments, and the uncertainty from such environments becomes the driving forces of the evolution. The component services can be dynamically integrated or detached from the system and thus the evolution is closely related to the architecture reconfigurations.

Having observed this, many works approach the problem from an architectural perspective, using architectural description languages to model the system structure and the dynamism [5]. However, there indeed exists a causal relationship between software architecture and the changing environments (including the surroundings and requirements, etc), but such a relationship is implicit. From end users' view, they care more about the running system's behaviors rather than the structure changes. Therefore, a modeling mechanism that can directly capture behaviors is more favorable. Secondly, in open environments, the software components are composed to form services (e.g., service composition), and this composition procedure is recursive. As a result, such systems naturally exhibit a hierarchical structure [6]. Moreover, time related properties are particularly relevant with the quality of the composite service, and in many cases, the violation of time constraints triggers the evolution process [7]. Timed automata have been proved to be tremendously useful for the verification of time related properties over the decades with good tool support [8]. Moreover, the transitions of timed automata can also be extended with probability distributions and this can be used to model the underlying uncertainty of involved systems [9]. However, the lack of high-level composable patterns for a hierarchical design hampers its further application. Users have to manually cast those terms into a set of clock variables with carefully calculated clock constraints, and the process is tedious and error prone [10].

The above considerations motivated us to propose a novel approach based on hierarchical timed automata to model and analyze dynamic evolution. Compared with existing works, our approach makes the following contributions: (1) The approach can directly support the modeling of temporal concerns, hierarchical structure, state transitions, and evolution behaviors. (2) We incorporate features of probabilistic transitions to model the uncertainties during operations characterized in open environments. By using the statistical model checking tools, our approach can support the verification of probability related properties. (3) We use both classical temporal logics as well as the probabilistic temporal logics to specify the properties that are supposed to be held, and these properties can be verified before, during, and after the dynamic evolution, respectively. (4) To render the model amenable to analysis by existing tool-sets, we also propose a translation algorithm to flatten the hierarchy. This paper builds upon our previous work [11], but with significant extensions. Particularly, the current paper complements the hierarchy flattening step with a formal translation algorithm and the attendant correctness proof as well as a time complexity analysis. We further revise and enhance the illustrative example taken from the online business domain as widely adopted in the literature [12–14] with more features—such as probabilistic transitions—and a more comprehensive set of experiments by scaling up the size. Moreover, a more rich set of properties to be verified has been presented to demonstrate the feasibility of our approach.

The rest of the paper is organized as follows. Section 2 introduces some theoretical background and the motivating example; Section 3 illustrates our approach; Section 4 presents the flattening algorithm and verification with performance evaluation; Section 5 discusses our approach and Section 6 compares our work. Section 7 concludes our paper.

2 Background

Timed automata are an extension of finite state automata to model real-time systems over time. Given the introduction of *clock*s, a run of the automaton along a sequence of consecutive transitions is of the form: $(l_0, \nu_0) \rightarrow (l_1, \nu_1) \cdots \rightarrow (l_p, \nu_p)$, where $(l_i)_{0 \leqslant i \leqslant p}$ denotes the location, $(\nu_i)_{0 \leqslant i \leqslant p}$ denotes the clock value and (l_i, ν_i) constitutes the state. Specifically, l_i is associated with some combinations of clock-related Boolean expressions to denote time invariants, marked as $Inv_{l_i}(\nu)$. Since time is continuous, timed automata theoretically have infinite states. By techniques such as region reduction, these infinite states can be equally reduced to finite states [8], and an interesting property of timed automata is that *reachability* is decidable.

Hierarchical timed automaton (HTA) introduces a refinement function to describe the hierarchy relationship between states. Therefore, it can naturally model the complex structures, for example, the composite states with several regions. Given the fact that in open environments, systems are usually constructed through sets of atom/composite services, the HTA intuitively corresponds to such complex hierarchical structures. To facilitate discussion, we have given related definitions of HTA formally, and then we will use an online order processing system as the motivating example to illustrate our approach.

Definition 1. Timed automaton can be defined as a tuple $\langle S, S_0, \sigma, C, Inv, M, T\rangle$, where S is a finite set of locations, S_0 is the finite set of initial locations, σ is a type function, and $\sigma : S \to \{\mathrm{BASIC}, \mathrm{COMPOSITE}, \mathrm{INIT}\}$, that is, σ returns a specific type of a location; C is a finite set of clocks, M is the set of events, which are two types: synchronized (coordinated by ! and ?) and asynchronized. $T \subseteq S \times (M \times \mathcal{CC} \times 2^C) \times S$ is a finite set of transition steps in which $\mathcal{CC}$ is a finite set of clock constraints and 2^C denotes the power set of reset clocks. For example, if an action a happens and the valuation of clocks satisfies the constraint g, it can cause a location s at time t to change to s'. This transition *trans* can be formalized as $\langle s, m, g, r, s'\rangle$ in which $m \in M$, and any clock, which is included within r, is reset and restarts from 0 as soon as the transition happens. For clarity, sometimes the transition step can also be written as $s \xrightarrow{m,g,r} s'$. For each transition $t \in T$, there are two associated functions, mapping to the related source and target locations respectively, i.e., $SRC : T \to S$, and $TGT : T \to S$.

Definition 2. Hierarchical timed automaton can be defined as a tuple $\langle F, E, \rho\rangle$, where F is a finite set of timed automata, E is the finite set of events, and ρ is a refinement function, mapping a location to a set of automata, i.e., $\rho : \bigcup_{A\in F} S_A \mapsto 2^F$ in which S_A denotes the location set of each element automaton in F. ρ constructs a tree (hierarchical) structure to the related automata.

Based on the refinement function, we can *recursively* define ρ^* to denote the set of descendent timed automata of a composite location s: $\rho^*(s) = \rho(s) \cup (\bigcup_{s_i\in S_{\rho^*(s)}} \rho(s_i))$. There is a special subset of descendent timed automata of a composite location s. For each element of such subsets, denoted by $\rho_{\mathrm{leaf}}(s)$, there are no further children timed automata, i.e., $\rho_{\mathrm{leaf}}(s) = \{A' | A' \in \rho^*(s) \wedge \forall s' \in A', \rho(s') = \emptyset\}$. If a transition t is involved with composite states, either the source or the target is composed of several orthogonal basic states. To model this, we utilize source and target restriction functions, which are defined as follows, $sr : t \to S$ where $S \subseteq \bigcup_{A_i\in\rho^*(SRC(t))} S_{A_i}$ and $tr : t \to T$ where $T \subseteq \bigcup_{A_i\in\rho^*(TGT(t))} S_{A_i}$. For both sets, elements are pairwise orthogonal locations if any. As their names suggest, these functions are mainly to restrict the transitions involved with those composite states, and they are yielding the actual sources or targets of a transition. Because of the composite location, it is possible that there are multiple active locations at the same time, we define the set of all the active locations as a *configuration*; and those active locations extended with the values of their corresponding clocks could be similarly defined as *timed configuration*. It can be regarded as a snapshot of the system.

In many cases, the software systems suffer from the dynamism and uncertainty in the environment [6], such as the Internet, and exhibit several kinds of random behaviors to certain degree. To enable the modeling of the underlying uncertainties, we extend our notion of timed automata and hierarchical timed automata with the ability to allow the probabilistic transitions as defined in the following.

Definition 3. Extended timed automaton can be defined as a tuple $\langle S, S_0, \sigma, C, Inv, M, Prob\rangle$, where S, S_0, σ, σ, C, and M are the same as those defined in the previous *timed automaton*. $Prob \subseteq S \times M \times \mathcal{CC} \times Dist(2^C \times S)$ is a finite set of probabilistic transition steps in which $Dist(2^C \times S)$ denotes the set of all probability distributions over $2^C \times S$, i.e., the set of all possible functions $\mu : (2^C \times S) \to [0, 1]$ such that $\Sigma_{q\in 2^C\times S}\mu(q) = 1$. This probabilistic transition *pt* can be formalized as $\langle s, m, g, \mu, s'\rangle$. Similarly, for each probabilistic transition $pt \in Prob$, there are two associated functions, mapping to the related source and target locations, respectively, i.e., $SRC : T \to S$, and $TGT : T \to S$. Differing from the definition

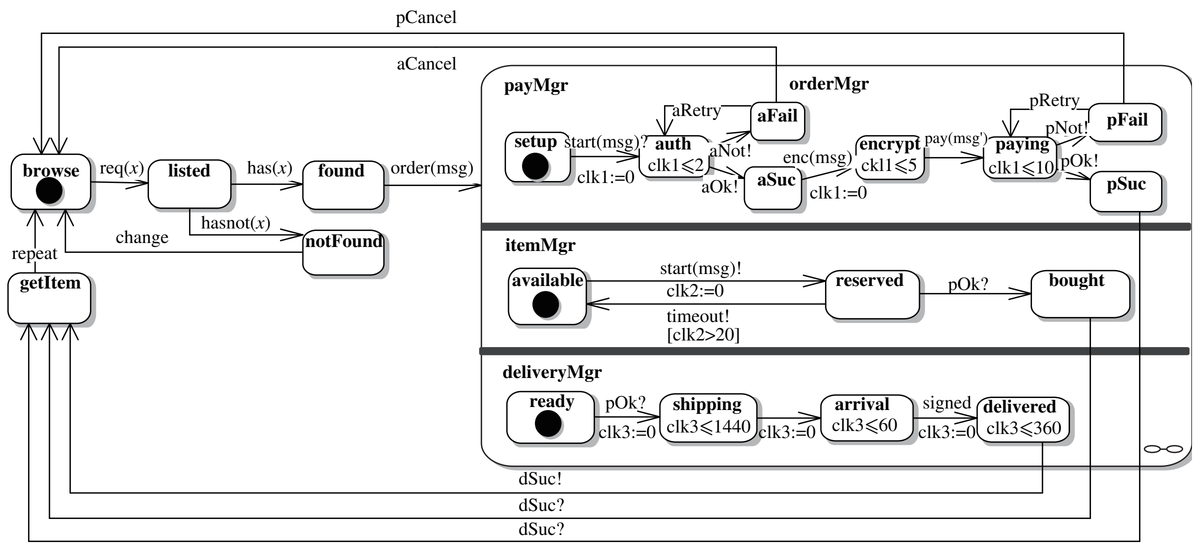


Figure 1 Example model illustration.

of probabilistic timed automaton given in [15], for simplicity, we do not allow the nondeterministic probabilistic transitions.

Definition 4. **Extended hierarchical timed automaton** can be similarly defined as a tuple $\langle eF, E, \rho\rangle$. The only difference from the previous notion of *hierarchical timed automaton* is that eF is the finite set of *extended timed automata*.

Statistical model checking is a very important approach of analyzing probabilistic models. Essentially it relies on finitely many runs of the stochastic model to get the samples, which further provides the statistical evidence for the satisfaction or violation of the specification based on hypothesis testing [16]. Differing from traditional model checking, it can be regarded as a tradeoff between testing and verification and can express quantitative properties of interest. Also differing from probabilistic model checking, it does not explore the whole state space, and thus can scale up to larger systems that may not be feasible for traditional numerical approaches.

3 Behavior modeling

In the sequel, we first present an overview of our approach, and then we use a concrete example taken from the e-business domain to illustrate our approach. Online order management is one of the most important activities in e-commerce. It involves the collaboration of multiple services, and also has strict temporal requirements. A complete e-commerce workflow typically consists of client-side service, and vendor-side service. On the vendor side, it further consists of payment management service, goods management service, and delivery management service. To reduce unnecessary complexity without sacrificing the integrity, we only focus on the vendor side, and simplify the client side service.

Generally, our approach consists of the following steps. First we leverage hierarchical timed automata to construct the system models under design. To cope with the uncertainty aspects, we leverage the extended hierarchical timed automata to model the probabilistic transitions. Particularly we model the system before, during, and after evolution, respectively. Second, we use temporal logics to specify the properties that are supposed to hold. Third, to verify against these properties, we use an indirect way to translate it to a set of sequential timed automata, i.e., we flatten the hierarchy so as to make it verifiable via existing model checkers. In this section, we mainly discuss the behavior modeling step, and the verification related steps are explained in the next section.

Figure 1 illustrates the behavioral modeling of the example. Clients browse the web site and send requests. The results indicate two branches. If the vendor has the item, the clients will order and three

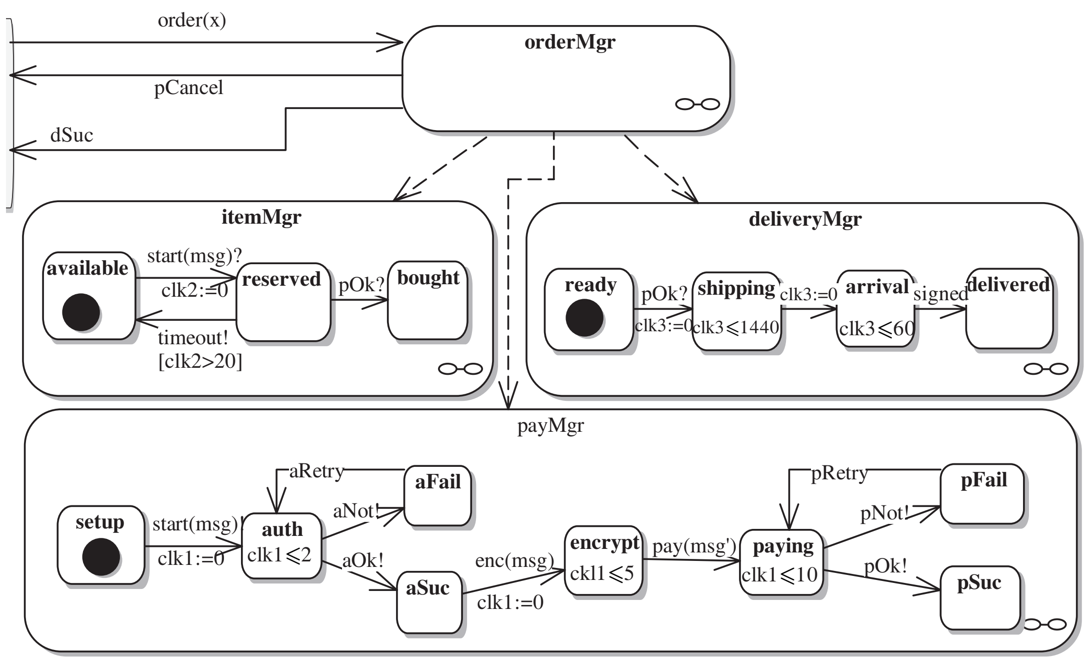


Figure 2 Hierarchy decomposition.

concurrent threads will execute, i.e., payment management (*payMgr*), item management (*itemMgr*), and delivery management (*deliveryMgr*). In payment management part, the process will start from *setup* state and authenticate the client's identity (*auth*). Normally, there is a time threshold for the authentication, and in the example, we set it 2 min. After passing the authentication, the following interactions will be encrypted, the time limit is set to be 5 min at most and the online paying step lasts at most 10 min as set in the example. For the item management aspect, once the item is ordered, it will be transferred to the reserved location and kept in that location for at most 20 min. After that period and no payment message received, it will be back to the *available* location again; otherwise, notified by the payment acknowledgement, it will be transferred to *bought* location. For the delivery management part, once the item has been paid successfully, the initial *ready* location will be transited to *shipping.* The attendant temporal constraints states that the item should finish shipping within one day (1440 min). Once arriving at the destination, it will be dispatched at most 1 h (60 min) and delivered to the clients within the next 6 h (360 min).

Figure 2 illustrates the partial hierarchy decomposition of the example. Since *orderMgr* is a composite location, based on the definition of refinement function, $\rho(orderMgr) = \{payMgr, itemMgr, deliveryMgr\}$.

However, as discussed previously, there are variable sources of uncertainty from the open environments. As Baresi et al. state, the world is open to new components that context changes could make dynamically available, and systems can discover and bind such components dynamically to the application while it's executing [17]. The service integrator might strengthen the security aspects, and want to add a new *sms* (short messages) authentication feature to the system on the fly. This requirement inevitably results in a dynamic evolution process. In *sms* authentication, it also has a strict time validity requirement, we can still use a clock variable *clk1* to model this concern. In the example, we set it to be no more than 8 min as time constraints ($clk1 \leqslant 8$). The evolved behavioral model is partially described in Figure 3.

Figures 1 and 3 describe the system models before (source) and after (target) the evolution respectively. System states should be handled carefully to keep the evolution ongoing in a safe and low-disruptive way. Kramer et al. firstly proposed the notion of *quiescent state* as a sufficient condition [18] for a node to be safely manipulated from a running system. Later, Vandewoude et al. [19] proposed an alternative mechanism: *tranquility.* Given a state S, if it is not currently engaged in a transaction that it initiated, or it will not initiate new transactions, or it is not actively processing a request, or none of its adjacent nodes

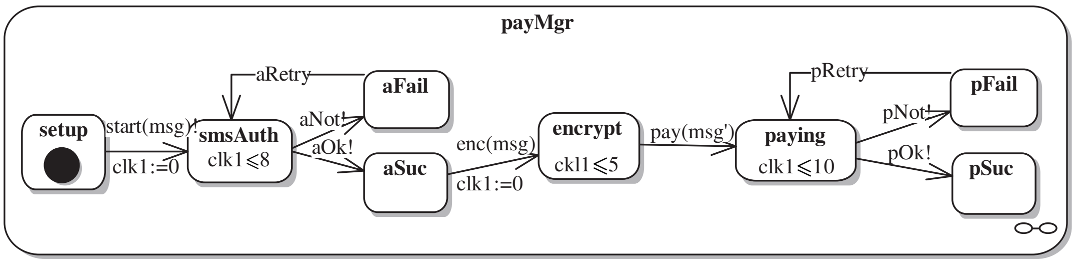


Figure 3 Evolved behavior model.

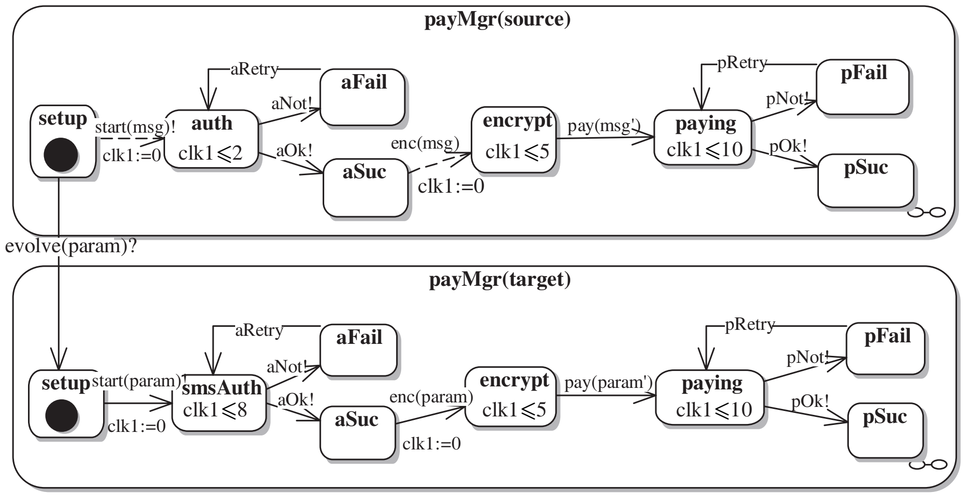


Figure 4 Behavioral model of evolution.

are engaged in a transaction in which it has both already participated and might still participate in the future, we can predicate the state is in a *tranquil* status. Compared with the previous one, *tranquility* has weaker requirements and causes lower-level disruption. In our paper, we adopt *tranquility* as the premise for the evolution, but our approach is not bound with any particular such mechanism. Given the definition of *tranquility*, we can impose the prerequisite for the dynamic evolution: the intersection of the active configuration set and evolution related element set is empty. Due to space limitations, we refer interested readers to [19] for details on the *tranquility* mechanism. In the example, the evolution related service is authentication service, so if the active configuration set contains any element of the related set, such as *auth,aFail,aSuc*, and it would be forbidden to enact the evolution; otherwise, it would be safe. As described in Figure 4, we take configuration $\langle\langle payMgr.setup\rangle, \langle itemMgr.available\rangle, \langle deliveryMgr.ready\rangle\rangle$ as an example to explain the evolution model. The evolution action is marked as *evolve*, and the parameters are passed through messages.

Since the evolution action does not involve *itemMgr* location or *deliveryMgr* location, the tuple element $\langle itemMgr.available\rangle$ and $\langle deliveryMgr.ready\rangle$ will not be affected by the evolution. For clarity, Figure 4 only displays the evolution related parts. Tranquility requires the interactions between replaced nodes and the environments be blocked until the end of the substitution process. Therefore, the related transition $setup \xrightarrow{start(msg)!,clk1:=0} auth$ and $aSuc \xrightarrow{enc(msg),clk1:=0} encrypt$ are marked as dashed lines, representing *restricted*.

The second type of evolution case involves the updated model of *sms* service. Because of the uncontrollable nature of the underlying environment, it is quite possible that the message sent is lost during transmission. To characterize this, we use probability as the parameters of the quality of the *sms* service. Therefore, this kind of evolution mainly deals with parameter adaptation [20]. To illustrate this, we elaborate on the *smsAuth* state, and make it a composite one but contains only one region. Figure 5 describes the activity. The inner behavior starts as soon as the process enters the composite state *smsAuth*. A short message is first sent (*smsSend*) to the customer within 2 min but without the guarantee

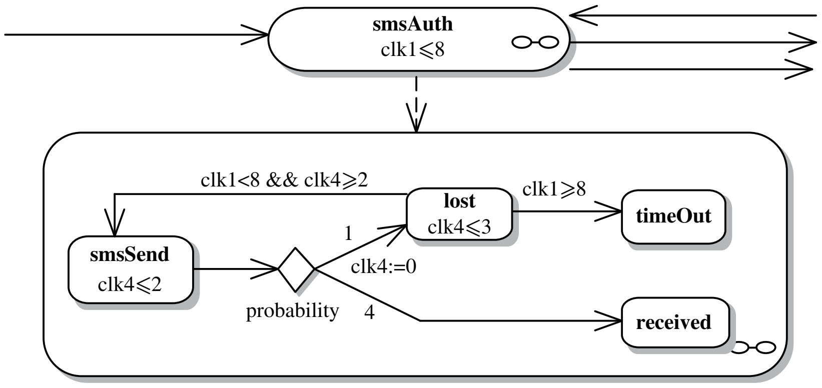


Figure 5 Hierarchy for smsAuth.

of delivery due to the uncertainty of quality of service or the underlying environment. A probability ratio is assigned to the successful and failed receipts of the message respectively based on preliminary knowledge or the experience from domain experts. In our example, we set the message loss (*lost*) possibility to be 1/5 and successful delivery to be 4/5 for default initial value. The diamond notation in Figure 5 represents the probabilistic choice. In case of message loss, it will repeat until being timeout (*timeOut*). Otherwise it will enter the *received* state. From Figure 5, we can derive the refinement relation, i.e., $\rho(smsAuth) = \{smsSend, lost, timeOut, received\}$.

4 Verification

In this section, we explain the verification of the evolution process based on the proposed models. The first kind of evolution involves the structural changes due to the replacement of existing components with different functional components. The verification has three stages, i.e., before the evolution, during the evolution and after the evolution. In each stage, we conducted the verification against the properties, and these properties are specified using computation tree logic (CTL). By *before evolution*, we mean the design artifact of the original system. By *after evolution*, we mean the updated models, which expose a different behavior sequence; while by *during evolution*, we mean the adaptation model connecting both the original and updated designs. As for the second kind of evolution, it mainly involves the parameter changes, aiming at non-functional properties. The model evolves without structural changes. Since the existing mainstream model checkers do not directly support the concept of hierarchy, to make it amenable for the direct analysis using available tool-sets, we first need to preprocess the model, including flattening the hierarchy and translating to a set of sequential timed automata.

4.1 Flattening algorithm

In our work, we use UPPAAL [21] (and its extension UPPAAL-SMC [22] for statistical model checking) as a test bed since its formalism is also based on timed automata, and this semantic similarity eases the verification process. As aforementioned, we need to flatten the hierarchy before verification. In UPPAAL, each timed automaton is termed as the *template*, and node as the *location*. We use UPPAAL's term in our translation and the flattening algorithm mainly has 5 steps. The pseudo-code is given in Algorithm 1.

1. For every timed automaton F in hierarchical timed automata M, which is not the root automaton, add a new *location*, and mark it as *inactive* (committed type[1)]).

2. For every transition t whose target is a composite state s, add the transitions from the *inactive* state to the default entry state inside s, the triggering event is the same as t's event. If the s itself is the entry state, the default entry state inside s is marked as entry state, otherwise, the state *inactive* is marked as the entry state.

1) In UPPAAL, a committed location cannot delay and the next transition must involve an outgoing edge of at least one of the committed locations.

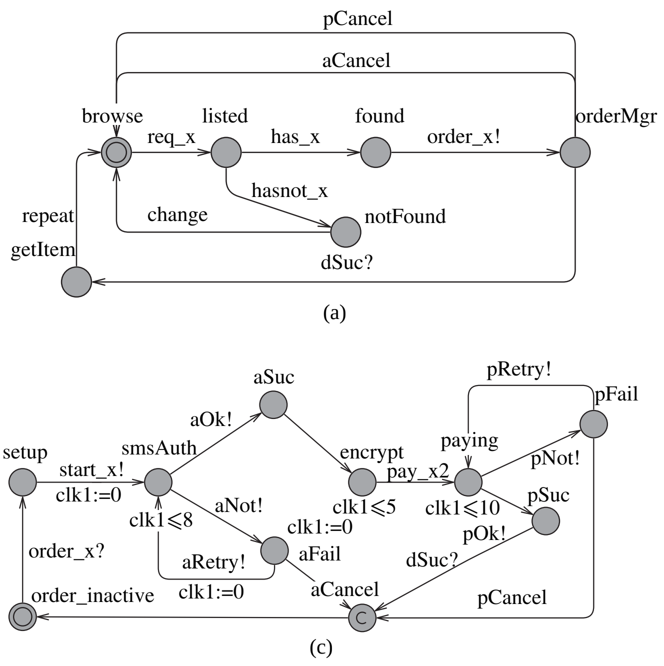


Figure 6 Corresponding UPPAAL models illustration. (a) orderMgr's corresponding model; (b) itemMgr's corresponding model; (c) payMgr's corresponding model; (d) deliveryMgr's corresponding model.

3. To synchronise the execution of multiple related automata, the event of the transitions whose targets are composite states is extended to the broad-cast channel, and marked as send type(!); meanwhile, in its children timed automata set, the transitions from *inactive* state to the default entry state are also extended to broad-cast channel, and marked as receive type(?).

4. For every non-root timed automata, add a special state (marked as *committed*), and add transitions from each element of the source restriction set of the ongoing transitions from the composite state to this special state, and extend this transition's event to the broad-cast channel, and marked as receive type(?).

5. Add the transitions from each special state resulted from the previous step to the *inactive* of the first step in the same timed automaton.

Algorithm 1 describes the steps of the translation. It contains four sequential loops. Assume M is the HTA model, and n is the number of total locations within M. The cost of each cycle in the first loop (Lines 3–7) is denoted by c_1. Then the cost of the first loop is $O(c_1 \times |M.F|)$. Since the theoretical maximum value of $|M.F|$ is n, we can deduce that the cost of the first loop is $O(c_1 \times n)$, i.e., $O(n)$. Similarly, we can calculate that the cost for the third loop (Lines 27–31) is also $O(n)$. As for the second loop (Lines 9–26), it further has three nested sub-loops (Lines 13–16, Lines 11–22, Lines 10–25, respectively). We first consider the most inner sub-loop (Lines 13–16), given a specific location s, the maximal number of transition targeted at is n, therefore, the corresponding cost is $O(n)$. The second inner sub-loop depends on the number of $|\rho(s)|$, and the maximal value is $n-1$. The outer sub-loop (Lines 10–25) depends on the number of $|A_i.S|$, and the maximal value is $n-1$. So the total cost for the second loop is $O(n^4)$. For the fourth loop (Lines 32–43), since the maximal number of $|t|$ is n^2, where $t \in \bigcup_{A_i \in M.F} \Sigma_{A_i}$, we get the corresponding cost for the fourth loop to be $O(n^4)$. So in total, the cost of Algorithm 1 is $O(n^4)$.

4.2 Correctness of translation

In [23], we have elaborated the formal semantics of hierarchical timed automata with multiform time extensions. Although in our context, we do not provide our model with features of multiform time. They are of the same nature, i.e., physical time. Nevertheless, our model can be regarded as a special degenerated case of the model proposed in [23], thus the defined semantics still hold. We also rely on the semantics of some model elements from UPPAAL. For example, *committed* type nodes. They are mainly related to the auxiliary locations introduced, and are used only to enable or disable transitions. They

Algorithm 1 Flattening algorithm

```
Data: M: Hierarchical timed automata models
Result: T: Set of timed automata models
 1: $T \longleftarrow \emptyset$; HashMap $map$
 2: for all $A_i \in M.F$ do
 3:    create a template $t$; add locations and transitions based on $A_i.S$ and $A_i.\Sigma$
 4:    if $A_i \neq A_{\mathrm{root}}$ then
 5:       add $A_i\_inactive$ location in $t$; mark $A_i\_inactive$ as committed
 6:    end if
 7:    $T \longleftarrow T \cup \{t\}$; map.add($A_i$,t)
 8: end for
 9: for all $A_i \in M.F$ do
10:    for all $s \in A_i.S \wedge \sigma(s) =$ COMPOSITE do
11:       for all $A_i \in \rho(s)$ do
12:          temp $\longleftarrow$ map.get($A_i$)
13:          for all in_tr $\in$ {incoming transitions to $s$} do
14:             add transition $trans$ from $A_i\_inactive$ to the entry location in temp
15:             augment the action of $trans$ with $broadcast$ channel
16:          end for
17:          if $s$ is initial then
18:             default entry location in $A_i$ is marked as initial in temp
19:          else
20:             $A_i\_inactive$ is marked as initial in temp
21:          end if
22:       end for
23:       map.update
24:       T.update
25:    end for
26: end for
27: for all $A_i \in M.F \wedge A_i \neq A_{\mathrm{root}}$ do
28:    temp $\longleftarrow$ map.get($A_i$)
29:    add join location $A_i\_join$ in temp
30:    mark $A_i\_join$ as committed
31: end for
32: for all $t \in \bigcup_{A_i \in M.F} \Sigma_{A_i}$ do
33:    for all $s \in sr(t) \wedge (\sigma(s) =$ COMPOSITE) do
34:       for all $A_j \in \rho^*(s)$ do
35:          temp = map.get($A_j$)
36:          add transitions $tt$ from each locations generated by $A_j.S$ to $A_j\_join$ in temp
37:          add transitions $tt'$ from $A_j\_join$ to $A_j\_inactive$
38:          associate $tt$ with channels, guards and clock resets based on $t$
39:          map.update
40:       end for
41:    end for
42:    $T \longleftarrow \bigcup_{A_i \in M.A}$map.get($A_i$)
43: end for
```

do not take time and thus are redundant for the configuration. We refer interested readers to [21] for details.

Based on the semantics, we use the notion of bi-simulation [24] to establish the behavioral equivalence between the hierarchical model and the set of flattened ones. For the two models to behave in the way that one simulates the other and vice versa, we have to demonstrate that there exists a binary relation R that both R and R^{-1} are simulations. Before proving the bi-similarity with respect to the reachability, we first introduce the following lemma.

Lemma 1. Let TAs be translated from the HTA by Algorithm 1, a hierarchical state (s, u) in HTA is reachable if and only if the corresponding state (s', u) is reachable in TAs.

Proof. We first consider the sufficiency direction.

1. Assume (s, u) is the entry state in HTA, according to the translation algorithm, regardless of whether it is in the root automaton or not, the corresponding state after translation is (s', u). Since the inserted inactive location is marked as committed and the transitions will take place without delay and reach the

default entry state in TAs. So, corresponding (s', u) is reachable in TAs.

2. Assume (s, u) is an intermediate non-entry state, it means there is at least one incoming transition. For such case, we use induction on the path length n to prove.

2.1 When $n = 0$, this degenerates to the case of default entry state, and from the above statements, we know it holds.

2.2 Assume when $n = k$ the conclusion holds, and there is a transition t targeting at (s, u), i.e., $(s_k, u_k) \xrightarrow{t} (s, u)$, which makes the path length equal to $k + 1$. According to the translation algorithm, if $\sigma(s) = \text{BASIC}$, the corresponding (s', u) is faithfully rendered by our algorithm and the same event of s can trigger the corresponding transition t' in the translated automata, i.e., $(s'_k, u_k) \xrightarrow{t'} (s', u)$. If $\sigma(s) = \text{COMPOSITE}$, according to the translation algorithm, we know that every composite state in the HTA model corresponds to several UPPAAL automata based on its refinement function ρ. But the transition t has been extended with the broadcast ability to synchronize these multiple automata and the corresponding state (s', u) can also be reached through the transition t', i.e., $(s'_k, u_k) \xrightarrow{t'} (s', u)$. In either possibility, the transition including the triggering events name, clock constraints are the same, and thus clock evaluations u carries over without changes. We can assert that there is a corresponding (s', u) reachable in the translated automata set (TAs).

For the necessity direction, we can analogously prove by induction on the length of path targeting at s and s_0 similar to the sufficiency direction introduced above.

Theorem 1. Let TAs be the translated set of UPPAAL automata from the HTA by Algorithm 1, the two models are bisimilar.

Proof. If we do not consider the notion of a *start* location, and the automata can be degenerated to a common labeled transition system (LTS). According to Lemma 1, given every state (s, u) in HTA, we have a corresponding state (s', u) reachable in TAs, we can establish an equivalence relation R, in order that $((s, u), (s', u)) \in R$ and based on the definition of R, we can say that (s', u) is equivalent to (s, u). Thus, straightforwardly, an active configuration C in the HTA is equivalent to an active configuration C' in TAs if for all $(s, u) \in C$ there is an equivalent $(s', u) \in C'$ and vice versa. Since those auxiliary locations added by our algorithm transit to other meaningful states immediately and will not be part of any configurations, we can also establish a converse relation R^{-1}, such that $((s', u), (s, u)) \in R^{-1}$. Given the relations R, R^{-1} and Lemma 1, we can conclude that the R is a bi-simulation of the derived LTS, and the two models are thus bisimilar.

Although Algorithm 1 targets the hierarchical timed automata, we can substitute that with the extended hierarchical timed automata, and then we get the translation algorithm for extended hierarchical timed automata. Since the only difference is in the probabilistic distribution over transitions, and we do not change the semantics of that during translation, thus similarly we can prove the correctness for the extended hierarchical timed automata. Due to space limitations, we do not include that in our paper. For simplicity, we do not utilize the full expression power of UPPAAL-SMC, as it supports the priced timed automata. Instead, our extended timed automata can be regarded as a special case of priced timed automata in which the clocks evolve at the same rate. Detailed discussion is beyond the scope of this paper, however for further details, please refer to [25].

4.3 Consistency verification

Once the hierarchical timed automata have been translated into a collection of concurrent timed automata, we can use existing model checkers to verify interested properties. As aforementioned, we mainly leverage UPPAAL (integrated with UPPAAL-SMC, ver. 4.1.18 with academic licence) due to the similarities of the underlying models. For the first type of evolution, the verification is conducted in three stages, i.e., before, during and after the evolution. The four example properties are categorized into two types: one is safety, and the other is liveness. For the second type, we use statistical properties to specify user's requirement and quantitative model checking to verify whether or not the model satisfies such kinds of properties.

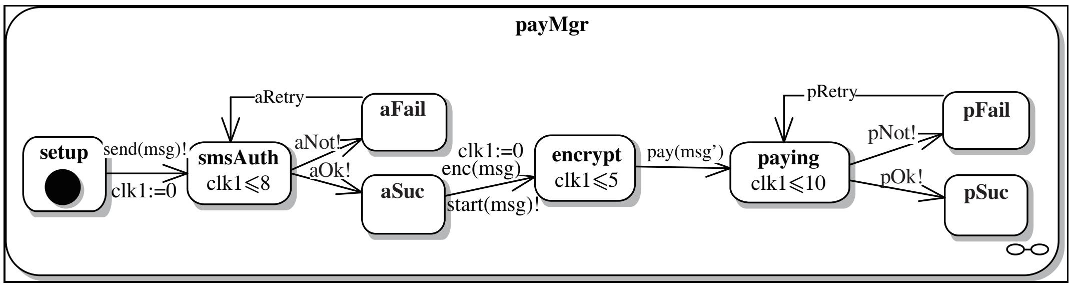


Figure 7 Revised design of payMgr.

For the safety category, regardless of whether the system evolves or not, when the customer pays for the item, it should not be in the *available* state. Put it in a CTL formula, we get P1 : $A\Box(\neg(payMgr.paying \wedge itemMgr.available))$; the other interesting property is that once the customer pays for an item successfully, the item will be no longer in the *available* state and in the delivery management, neither will it stay in the *ready* state again. Then similarly, we get P2 : $A\Box(\neg(payMgr.pSuc \wedge (itemMgr.available \vee deliveryMgr.ready)))$. As to the liveness category, one property that the system should hold is: as long as an requested item is found, the item will be either *bought* or *available*, but not in the *reserved* state. In CTL formula, we get P3 : $A\diamond(found \rightarrow (itemMgr.bought \vee itemMgr.available))$; as another example of such kind, once the customer pays successfully, eventually, the item will be delivered. This property can be specified as follows: P4 : $A\diamond(orderMgr.pSuc \rightarrow deliveryMgr.delivered)$.

Based on the flattening algorithm in the previous section, we translate the system before the evolution into the four concurrent automata recognized by UPPAAL as partially illustrated by Figure 6. Given the model and the specification, we verified the design and found that it satisfied all the above aforementioned properties.

We continued to verify the system after evolution against the interested properties. The result is that the system satisfies the third and the fourth property but not the first or the second. By inspecting the counter example, we realized that the time limit of the new *sms* authentication module is too long. Since by combining the time spent on the encryption and the payment, it is possible that before the timeout of payment, the reservation of the item will expire and roll back to the *available* state. Therefore, it is possible that the customer has paid successfully but the item's status is still available, leading to violation of P1 and P2 (i.e., $payMgr.paying \wedge itemMgr.available$ and $payMgr.pSuc \wedge itemMgr.available$). Moreover, once the system is in such a configuration, since the channel of *payMgr.pSuc* could not synchronize with the item, this will lead to a deadlock situation. In every individual step, the time requirement is fine per se, but when put together, inconsistency would happen. We investigated our original design given the traces generated by the counter example and found that changing the item's state to *reserved* before passing the authentication is not reasonable. Consequently, we revised our design and postponed the transition to the *reserved* state after passing the authentication. The reason why the first verification failed to disclose such a design flaw is that before evolution, the original authentication time plus the encryption time is still less than the predefined threshold. The revised design is given in Figure 7, which only includes the changed part. The transition from *aSuc* to *encrypt* has been renamed to *start(msg)!*, and thus the transition from *setup* to *smsAuth* has been renamed. We again model checked the revised design against the four properties and this time, all of them were satisfied.

The last step is to verify the evolution phase. Since we revised the model as described in Figure 7, the updated behavioral models are instead translated to the UPPAAL's input templates. Again, we model checked the design against the four properties above and concluded that all of them hold. That is, the configuration including $\langle\langle payMgr.setup\rangle, \langle itemMgr.available\rangle, \langle deliveryMgr.ready\rangle\rangle$ behaves consistently with the required safety and liveness properties. Figure 8 illustrates the UPPAAL models for the evolution phase. Since the tranquility mechanism requires that the authentication services should be inactive so as to be replaced, the related nodes and transitions are deactivated accordingly. Service information before evolution, such as user data, is transmitted to the target nodes as messages.

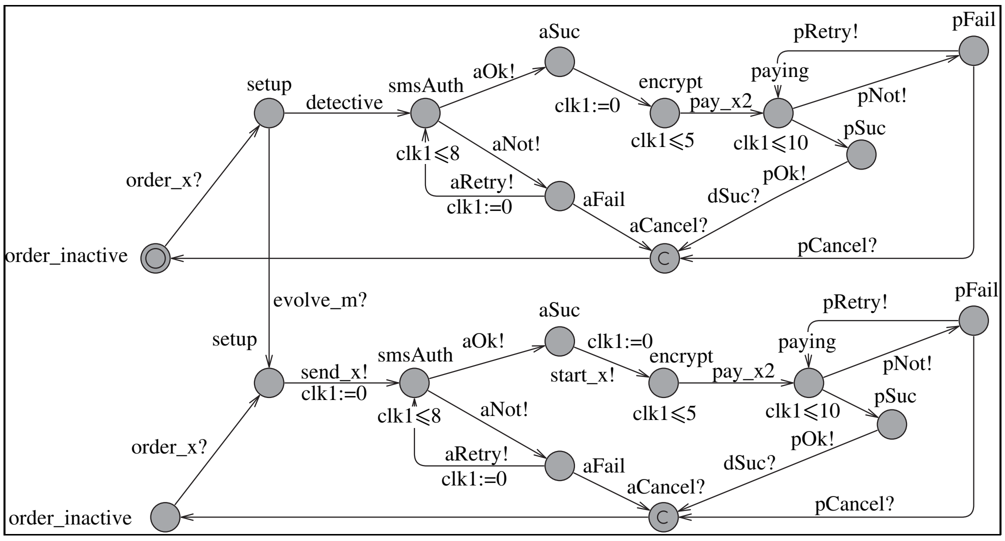


Figure 8 Evolution model illustration.

Table 1 Summary of the experiment results (N = number of items, R/V = Residential/Virtual Memory, NC = Not Concluded, OM = Out of Memory)

N	Before evolution		During evolution		After evolution		During evolution		After evolution	
	States	R/V (MB)	States	R/V (MB)	States	R/V (MB)	(Revised) States	R/V (MB)	(Revised) States	R/V (MB)
1: P1	1158	7.4/30.5	22	3.2/21.2	29	7.4/30.5	2952	7.5/30.6	1152	7.4/30,5
1: P2	1230	7.4/30.5	24	3.6/22.4	41	7.5/30.5	3648	7.6/30.9	1220	7.5/30.6
1: P3	1270	7.6/30.9	1828	3.2/29.5	1398	7.6/31.0	3810	7.8/31.4	1254	7.6/31.0
1: P4	1294	7.6/31.1	2034	3.5/29.0	1454	7.7/31.1	4032	8.0/31.5	1284	7.7/31.1
6: P1	182646	17.1/49.6	27	7.7/55.6	65	7.5/30.6	391320	28.7/71.9	155440	15.9/47.0
6: P2	182974	17.3/50.0	34	8.2/53.3	206	7.6/31.0	467280	28.9/72.2	155764	16.0/47.3
6: P3	183634	17.5/50.4	642268	8.9/61.8	626962	41.4/96.6	469922	29.3/73.0	156294	16.3/47.8
6: P4	249054	18.4/52.0	1316274	8.6/62.0	956902	46.0/105.9	732496	31.1/76.6	221844	17.0/49.4
10: P1	10496886	573.0/1146.7	31	7.6/30.8	111	7.5/30.7	25626960	1252.5/2519.2	8542000	484.2/969.1
10: P2	10497214	574.6/1149.8	42	7.8/31.3	508	7.8/31.2	25626960	1256.1/2526.3	8542324	485.9/972.6
10: P3	10507474	576.3/1153.2	NC	OM	NC	OM	25668002	1259.0/2532.1	8550534	487.1/974.8
10: P4	12070622	585.4/1171.5	NC	OM	NC	OM	31921488	1284.5/2583.6	10115732	497.0/994.9

To scale up the size of our example, we use the number of repository items as the parameter (denoted by N). We measured the number of generated states and the consumption of memory needed to perform the complete verification. The experiments are conducted on a PC with an Intel i7-3770 3.4 GHz processor and 4.0 GB RAM running 64-bit Windows 7 Professional Operating System. The versions of UPPAAL (*verifyta.exe*) and jdk are 4.1.18 with academic licence and 1.8.0, respectively. The complete experiment data (i.e., before, during and after the evolution) gathered are given in Table 1. Besides, after revising our original design, we re-conducted the experiments and these data results are also given in Table 1.

For Properties P1 and P2 in the original design, since they do not hold, and UPPAAL can find a counter example without exploring the whole state space, the performance data collected (i.e., states, memory consumption and time) in Table 1 are only those explored before finding the counter example. We also conducted experiments on the evolution model with the original design, since the properties in our case study are global ones. Similarly, the same properties do not hold and the results are only those explored before finding the counter example. Since we only revised the evolution design, thus the models before evolution will not be affected. After revision of the evolved model, all the four properties now hold.

As the size goes up, we find that soon the generated state space grows beyond the machine's capability and we use OM (out of memory) and NC (not concluded) in the table to describe this phenomenon. Based on our analysis in Subsection 4.1, for our translation algorithm, the time cost is still polynomial to the number of locations (n) in HTA, i.e., $O(n^4)$, and the maximal number of auxiliary nodes is linear to n. Thus the main reason for the state explosion is introduced by the parallel composition of timed automata. In Table 1, we can find that even for the same start system, the number of states explored is usually different for different properties. This is due to the optimization strategies utilized by UPPAAL. Generally, during the evolution, since it involves two versions of the models, the number of states are much more than the other two stages. This can be observed in Table 1. Theoretically, timed automaton has infinite number of states because of the dense representation of time. By using the notions of region reduction, the infinite becomes finite, making the state exploration possible for model checking [8]. For details of state space generation and exploration techniques in UPPAAL, please refer to [21].

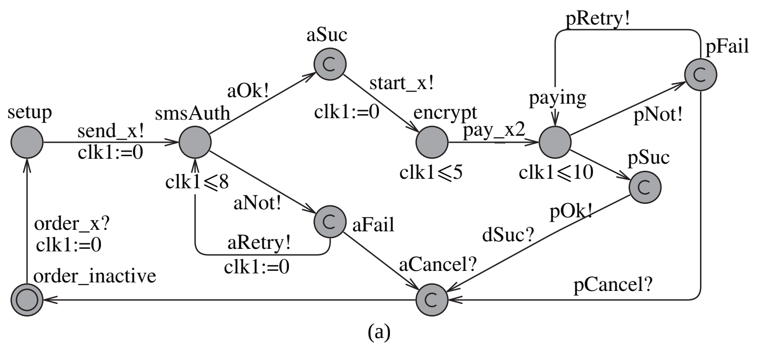
As to the second type of evolution, users might be interested that in spite of the uncertainty of message loss, given a period of time (e.g., 8 time minutes in *smsAuth*), the probability of successful delivery of message should be above a certain threshold, for example, more than 96%. Following UPPAAL's syntax for probabilistic temporal logics, we get P5: $Pr[< 8](\diamond smsAuth.recived) \geqslant 96\%$. Based on the composition hierarchy given in Figure 5 and the translation algorithm, the resulted flattened UPPAAL models are described by Figure 9. To make it workable for UPPAAL SMC, we still need to moderately revise the original payMgr model. As can be observed in Figure 9(a), the location type of *aSuc, aFail, pSuc,* and *pFail* has been changed to *committed* type since it will transit instantaneously without time cost; while in Figure 9(b), those locations with time invariants will enable the transition stochastically. Particularly, the transitions take place according to uniform distribution [22]. Therefore, we could use the given parameters of the model (design-time estimates) to conduct the statistical verification.

The curve in Figure 10(a) illustrates the cumulative probability distribution for the successful delivery of a message as time goes by based on the initial parameters. Given the initial estimate for the successful message delivery being 0.8, it is clear that the property P5 is satisfied. However, as argued in [20], the estimates are seldom correct. The problems is that these actual parameters may change over time. In running phase, real-time values are needed to update the estimates in order to make the model faithfully reflect real behaviors and thus be useful for analysis or prediction. If we have external gauges in the implementation of the system recording the message loss times and success delivery times, respectively, we could update the model parameters and thus verify whether the model continues satisfying the requirement more accurately. We use simulations to illustrate this. For example, after a certain period of running, the data collected tell that the initial value for message loss are underestimated, and the actual value should be up to 1/3. Based on this new ratio value, the model checker finds that P5 does not hold any longer. Figure 10(b) illustrates the updated cumulative probability distribution for successful message delivery over time. The cumulative probability given the time limit is only approximately 91%. Once the violation of the requirement is detected, the system could evolve by replacing the component service, i.e., *smsAuth* in our example, with one of higher reliability for message delivery until the property is satisfied.

For the experiments to verify Property P5, we set the values of false negative (α) probability, false positive probability (β) and uncertainty probability to be all 0.001. Thus it can give a relative accurate result with the confidence rate to be 0.999.

5 Discussion

As described in Section 4, the states in our approach do not mean fine-grained, algorithm-level data values. Instead, they denote the coarse-grained, interface-level, observable information. This is due to the characteristics of open environments. Indeed, in such environments, the system integrators are not necessarily the component developers, and details of intra-component level states are hidden from the end-users. We also employ the concept of *tranquility* as the basis for the evolution. However, *tranquility*



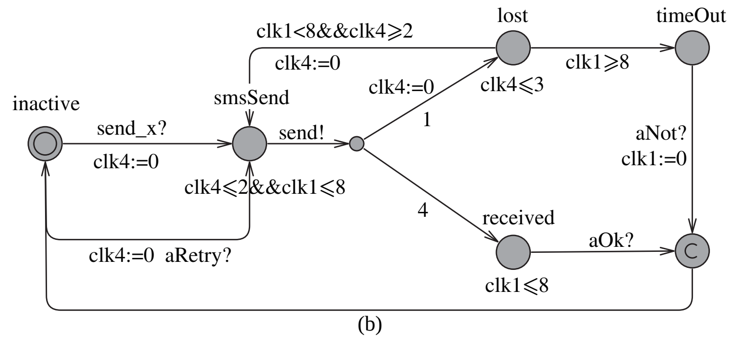
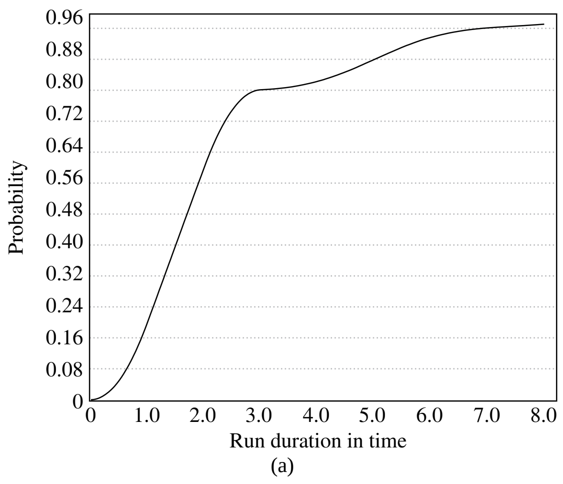


Figure 9 Flattened UPPAAL models for the two synchronized automata: payMgr and smsAuth. (a) payMgr's corresponding model; (b) smsAuth's corresponding model (with extended probability).



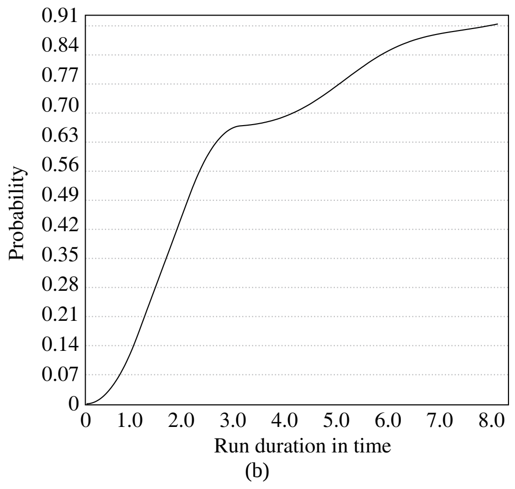


Figure 10 Cumulative probability for successful message delivery given different parameters. (a) P5 passes with initial parameters; (b) P5 fails with updated parameters.

is not free and requires some assumptions from the environments. Although in our example, this is trivial as to the verification result, since the timing of *itemMgr*'s *reserved* state is triggered after passing the authentication. Statistics indicate that only at very rare situations, the tranquility conditions are not reachable. Detailed explanations are presented in [19].

Hierarchical timed automata actually formalize a tree-like structure of the modeled system. It's partially motivated by the UML state machine diagram [26], in which the composite state imposes the hierarchy. However, involving composite locations, there are some transitions that directly point to or start from arbitrarily inner locations in composite ones. In UML superstructure specification, it states that if a transition terminates on an enclosing state and the enclosed regions do not have an initial pseudo-state, the interpretation of this situation is a semantic variation point. In some interpretations, this is considered an ill-formed model [26]. Therefore, in our model, to avoid potential ambiguity, we do not consider such transitions. The approach holds an implicit assumption that normally such inter-level transitions will not take place between sub-locations in different composite ones, but of course, they can be synchronized through channels. That's the potential limitation of our approach.

In the motivating example, messages pass between different states to transmit related information, for example, the item number and user identity. However, in UPPAAL, it does not support the message passing. We use an indirect way to get around the problem. Since the number of messages is finite and we encode these messages to different variables. During the transition, we manipulate the values of these variables and these values exactly correspond to the predefined messages. Verification is a broad issue, in the paper we emphasize on the consistency aspect. Thus we did not address the service substitution assurance mechanisms [27]. In our approach, it is probable that there are multiple evolution strategies, each of which has different set of source states, target states and tranquility status. Despite the complexity it introduces to the analysis, on one hand, we gain a high level of flexibility; on the other, we can inspect the state transition traces of the system, which cannot be realized through traditional structure based models. In our current work of the extended timed automata, since we rely on UPPAAL-SMC for verification, it supports limited number of stochastic models. For bounded delay in locations with invariants, uniform distribution is assumed; while for unbounded delay, exponential distributions with user-defined rates are used [22]. That's also the potential limitation of our work.

6 Related work

In [28], Huang et al. exploited the reflection mechanism to establish the causal relationship between the requirement/environment and the software artifact. In [29], Ma et al. leveraged inheritance and polymorphism to enact dynamic evolution. These work do not support formal consistency analysis. The work on model-based verification include [30–33]. In [30], Baresi et al. studied the domain-specific features of publish-subscribe style-compliant software architectural styles and proposed customized state-generation algorithms, which effectively reduced the state space. Their work mainly verified whether the specific software architecture satisfies the design-level requirements; while ours mainly concentrate on the behavioral consistency during evolution. In [32], Hayden et al. proposed a merge-based approach for enabling dynamic evolution. Two versions of the program, i.e., the old and the evolved, are bound and specified. Therefore the consistency can be checked against based on the specification; a similar work includes [31]. They rely on specific language and corresponding compilers. This assumption usually does not hold in open environments. In [33], Zhang et al. tried to use Petri-Net to model the components' behavior, and LTL formulas to model specification, thus can utilize existing tools to model check the interested properties. However, the approach neither addressed the temporal aspects of the system, nor the hierarchical structures.

In [34], Zhang et al. proposed a tool chain to support the modeling of behaviors by property sequence chart. The tools complement the UML sequence diagrams with rigorous semantics and feature the modeling ability of time and probability for service compositions. However, the work emphases extending the traditional scenario-based modeling language to model service compositions and does not address the dynamic evolution aspect. In [35], David et al. leveraged the expressive power of probabilistic timed automata to model the cyber-physical systems under varying environmental settings. In [36], Xu et al. studied the problem of detecting context inconsistency for Internetware computing. The proposed approach could efficiently detect contextual noises. However, these works did not address the evolution concerns either. In [9], Calinescu et al. argued the importance of applying quantitative verification techniques for dynamic evolvable systems in open environments. In line with this, they proposed a framework for the development of adaptive service-based systems [37]. Different from our approach, they only addressed the evolution of QoS aspects, without considering the structural changes.

There have also been other proposals based on graph rewriting techniques (e.g., [38, 39]). In [38], Hölscher et al. presented a semantics for UML based on the translation of a given model into a graph transformation system. The graph transformation system comprises transformation rules and a working graph representing current system states. But these states are limited to simple states instead of composite ones as supported in our approach. Xu et al. [39] used attributed graph grammar to model the system and graph transformation to model the dynamic evolution. Graph based notation has the visual intuitiveness advantage. However, when the model scales up, the cost of verification grows quickly, due to the fact that the matching are implemented through graph morphism. Besides, all the above mentioned approaches of such category lack a direct support for the modeling and analysis of dynamic evolution.

7 Conclusion

In this paper, we proposed a hierarchical timed automata based approach to model and analyze the dynamic software evolution from a behavioral perspective. Specifically, our work can support the analysis of both functional evolution (with structural changes) and non-functional evolution (with parameter changes). To the best of our knowledge, this is the first approach employing hierarchical timed automata to model evolution process through behavioral perspective. A motivating example with a comprehensive set of performance evaluations is discussed to illustrate the feasibility of our approach.

Despite the fact that our approach mainly targets the design phase, it can be also applied in the running phase since the boundary between different phases is now already blurring. For example, the model can be kept alive and verified during runtime. This is particular important for parameter level evolution, since the actual data to update the estimates can only be collected at runtime. In this paper, we used the notion *tranquility* as the basis for the dynamic evolution. However, the approach is not bound by the mechanism and is open to other alternatives, for example [40], but the integration of these more fine-grained mechanisms is left to future work.

Acknowledgements This work was partially funded by National Natural Science Foundation of China (Grant Nos. 61202002, 61202097, 61379157), Collaborative Innovation Center of Novel Software Technology and Industrialization, Chinese Fundamental Research Funds for the Central Universities (Grant No. NZ2013306), National Basic Research Program of China (973 Program) (Grant No. 2014CB744905), and Natural Science Foundation of Jiangsu Province (Grant No. BK20131277). The authors would like to thank Prof. Zining CAO, Dr. Taolue CHEN, Prof. Dehui DU, Jiayi GU, and Yankai HUANG for their help with the work.

Conflict of interest The authors declare that they have no conflict of interest.

References

1 Yang F, Lü J, Mei H. Technical framework for Internetware: an architecture centric approach. Sci China Ser-F: Inf Sci, 2008, 51: 610–622
2 Wang H, Wu W, Mao X, et al. Growing construction and adaptive evolution of complex software system (in Chinese). Sci Sin Inform, 2014, 44: 743–761
3 Fu J M, Tao F, Wang D, et al. Software behavior model based on system objects. J Softw, 2011, 22: 2716–2728
4 Wang Q X, Shen J R, Wang X, et al. A component-based approach to online software evolution. J Softw Maint Evol-Res Pract, 2006, 18: 181–205
5 Oreizy P, Medvidovic N, Taylor R. Runtime software adaptation: framework, approaches, and styles. In: Companion of the 30th International Conference on Software Engineering, Leipzig, 2008. 899–910
6 Lü J, Ma X X, Tao X P, et al. On environment-driven software model for Internetware. Sci China Ser-F: Inf Sci, 2008, 51: 683–721
7 Kazhamiakin R, Pandya P, Pistore M. Timed modelling and analysis in web service compositions. In: Proceedings of 1st International Conference on Availability, Reliability and Security, Vienna, 2006. 840–846
8 Alur R, Dill D. A theory of timed automata. Theor Comput Sci, 1994, 126: 183–235
9 Calinescu R, Ghezzi C, Kwiatkowska M, et al. Self-adaptive software needs quantitative verification at runtime. Commun ACM, 2012, 55: 69–77
10 Dong J S, Hao P, Qin S C, et al. Timed automata patterns. IEEE Trans Softw Eng, 2008, 34: 844–859
11 Zhou Y, Ge J D, Zhang P C. Hierarchical timed automata based verification of dynamic evolution process in open environments. In: Proceedings of the International Conference on Software and System Process, Nanjing, 2014. 144–148
12 Song W, Tang J H, Zhang G X, et al. Substitutability analysis of WS-BPEL services (in Chinese). Sci Sin Inform, 2012, 42: 264–279

13 Zeng J, Sun H L, Liu X D, et al. Dynamic evolution mechanism for trustworthy software based on service composition. J Softw, 2010, 21: 261–276

14 Zhou Y, Ma X X, Gall H. A middleware platform for the dynamic evolution of distributed component-based systems. Computing, 2014, 96: 725–747

15 Hartmanns A, Hermanns H. A modest approach to checking probabilistic timed automata. In: Proceedings of 6th International Conference on Quantitative Evaluation of Systems, Budapest, 2009. 187–196

16 Legay A, Delahaye B, Bensalem S. Statistical model checking: an overview. In: Proceedings of First International Conference on Runtime Verification, St. Julians, 2010. 122–135

17 Baresi L, Di Nitto E, Ghezzi C. Toward open-world software: issue and challenges. Computer, 2006, 39: 36–43

18 Kramer J, Magee J. The evolving philosophers problem: dynamic change management. IEEE Trans Softw Eng, 1990, 16: 1293–1306

19 Vandewoude Y, Ebraert P, Berbers Y, et al. Tranquility: a low disruptive alternative to quiescence for ensuring safe dynamic updates. IEEE Trans Softw Eng, 2007, 33: 856–868

20 Epifani I, Ghezzi C, Mirandola R, et al. Model evolution by run-time parameter adaptation. In: Proceedings of 31st International Conference on Software Engineering, Vancouver, 2009. 111–121

21 Behrmann G, David A, Larsen K. A tutorial on Uppaal. In: Proceedings of Formal Methods for the Design of Real-Time Systems, Bertinoro, 2004. 200–236

22 David A, Larsen K, Legay A, et al. Uppaal SMC tutorial. Int J Softw Tools Technol Transfer, 2015, 17: 397–415

23 Zhou Y, Baresi L, Rossi M. Towards a formal semantics for UML/MARTE state machines based on hierarchical timed automata. J Comput Sci Technol, 2013, 28: 188–202

24 Milner R. Communicating and Mobile Systems: the Pi Calculus. Cambridge: Cambridge University Press, 1999

25 Behrmann G, Larsen K, Rasmussen J. Priced timed automata: algorithms and applications. In: Proceedings of International Symposium on Formal Methods for Components and Objects, Amsterdam, 2005. 162–182

26 OMG. Specification. Unified Modeling Language: Superstructure Version 2.2. OMG Formal Document, 2009

27 Cavallaro L, Di Nitto E, Pradella M. An automatic approach to enable replacement of conversational services. In: Proceedings of the International Conference on Service-Oriented Computing, Stockholm, 2009. 159–174

28 Huang G, Mei H, Yang F Q. Runtime software architecture based on reflective middleware. Sci China Ser-F: Inf Sci, 2004, 47: 555–576

29 Ma X X, Zhou Y, Pan J, et al. Constructing self-adaptive systems with polymorphic software architecture. In: Proceedings of International Conference on Software Engineering and Knowledge Engineering, Boston, 2007. 2–8

30 Baresi L, Ghezzi C, Mottola L. Loupe: verifying publish-subscribe architectures with a magnifying lens. IEEE Trans Softw Eng, 2011, 37: 228–246

31 Chen H B, Yu J, Hang C Q, et al. Dynamic software updating using a relaxed consistency model. IEEE Trans Softw Eng, 2011, 37: 679–694

32 Hayden C, Magill S, Hicks M, et al. Specifying and verifying the correctness of dynamic software updates. In: Proceedings of International Confernece on Verified Software: Theories, Tools, Experiments. Berlin: Springer, 2012. 278–293

33 Zhang J, Cheng B. Model-based development of dynamically adaptive software. In: Proceedings of 28th International Conference on Software Engineering, Shanghai, 2006. 371–380

34 Zhang P C, Leung H, Li W R, et al. Web services property sequence chart monitor: a tool chain for monitoring BPEL-based web service composition with scenario-based specifications. IET Softw, 2013, 7: 222–248

35 David A, Du D, Larsen K, et al. An evaluation framework for energy aware buildings using statistical model checking. Sci China Inf Sci, 2012, 55: 2694–2707

36 Xu C, Liu Y P, Cheung S C, et al. Towards context consistnecy by concurrent checking for Internetware applications. Sci China Inf Sci, 2013, 56: 082105

37 Calinescu R, Grunske L, Kwiatkowska M, et al. Dynamic QoS management and optimization in service-based systems. IEEE Trans Softw Eng, 2011, 37: 387–409

38 Hölscher K, Ziemann P, Gogolla M. On translating UML models into graph transformation systems. J Vis Lang Comput, 2006, 17: 78–105

39 Xu H Z, Zeng G S, Chen B. Conditional hypergraph grammars and its analysis of dynamic evolution of software architectures. J Softw, 2011, 22: 1210–1223

40 Ma X X, Baresi L, Ghezzi C, et al. Version-consistent dynamic reconfiguration of component-based distributed systems. In: Proceedings of 19th Symposium on Foundations of Software Engineering, Hungary, 2011. 245–255