Formal Analysis of Publish-Subscribe Systems by Probabilistic Timed Automata

Fei He[1], Luciano Baresi[2], Carlo Ghezzi[2], and Paola Spoletini[2]

[1] Department of Computer Science & Technology, Tsinghua University
Beijing, China, 100084
hef02@mails.tsinghua.edu.cn
[2] Dipartimento di Elettronica e Informazione, Politecnico di Milano
Milano, Italy, 20133
{baresi,ghezzi,spoleti}@elet.polimi.it

Abstract. The publish-subscribe architectural style has recently emerged as a promising approach to tackle the dynamism of modern distributed applications. The correctness of these applications does not only depend on the behavior of each component in isolation, but the interactions among components and the delivery infrastructure play key roles. This paper presents the first results on considering the validation of these applications in a probabilistic setting. We use probabilistic model checking techniques on stochastic models to tackle the uncertainty that is embedded in these systems. The communication infrastructure (i.e., the transmission channels and the publish-subscribe middleware) are modeled directly by means of probabilistic timed automata. Application components are modeled by using statechart diagrams and then translated into probabilistic timed automata. The main elements of the approach are described through an example.

1 Introduction

The publish-subscribe architectural style [1–3] has recently emerged as a promising approach to tackle the dynamism and flexibility of modern distributed applications. Components do not communicate directly, but their interactions are mediated by a dedicated element called *dispatcher*. Components dynamically *subscribe* to the messages they are interested in, and the dispatcher *notifies* them as soon as a message that matches their subscriptions is *published* by one of the other components. The dispatcher is the only element that knows how to route the messages in the system, and thus the sender of a message does not know its receivers. This peculiarity allows components to join and leave an application seamlessly without any need to restructure the whole system.

The correctness of these applications does not only depend on the correct behavior of each component in isolation. We also need the right intertwining among subscriptions, publications, and notifications, to allow components to receive the messages they need, and an infrastructure that actually delivers all the messages exchanged within the system. Therefore, the formal analysis of publish and subscribe systems must consider two orthogonal aspects: (a) the subscriptions and unsubscriptions that can dynamically change the topology of the system, and its interaction paths, and (b) the underlying infrastructure that cannot always guarantee that all messages, along with subscriptions and unsubscriptions, be delivered to all interested parties.

Among the many attempts [4–10] to model and validate publish-subscribe systems, model checking has been considered as an attractive solution. However, all these works assume deterministic systems and neglect the uncertainty that characterize practical publish-subscribe applications. Differently from these works, in this paper we do not discuss the fine-grained analysis of subscriptions and notifications, but we present the first results on extending these approaches by considering the problem in a probabilistic setting. We use probabilistic model checking techniques on stochastic models to tackle the uncertainty that is intrinsic in these systems. Even if we assume that application components are correct, stochastic models help us reason on the reliability of the infrastructure, that is, the probability with which messages can be lost during the transmission. Moreover, since probabilistic model checking involves the exhaustive exploration of all possible paths, it can also supply important information about the model: for example, the optimal size of buffers, average delays, and so on.

In our approach, we provide independent models for the communication infrastructure, which includes the transmission channels and the middleware, and for application components. The transmission channels provide the mechanisms for message delivery. The middleware supports subscriptions, unsubscriptions, and message routing. Both are problem-independent, and their models are reusable in different systems. Application components are problem-specific and are not usually reused. In this paper, the communication infrastructure is modeled by means of probabilistic timed automata directly. Application components are modeled with statechart diagrams, and we provide some easy clues for translating these diagrams to probabilistic timed automata. An example helps us validate the effectiveness of our approach by means of the probabilistic model checker PRISM.

The rest of the paper is organized as follows. Section 2 briefly surveys some related works. Section 3 introduces an abstract model of publish-subscribe systems. Section 4 provides some background definitions about probabilistic timed automata. Section 5 presents our proposal, while Section 6 exemplifies it on a simple case study. Section 7 concludes this paper.

2 Related Work

This paper builds on the previous efforts of some of the authors [4–6] by extending the formal analysis of publish-subscribe systems in a probabilistic setting. In [4], application components are modeled as UML statechart diagrams while the communication infrastructure is supplied as a configurable predefined component. UML statechart diagrams are translated into Promela and validated through the SPIN model checker. While this approach builds on top of an existing model checker, [5,6] extend the Bogor model checker and the communication mechanisms of publish-subscribe infrastructures are embedded in it. Some domain-specific knowledge is used to reduce the state space.

In [7,8], the authors present a generic framework for automatically analyzing publish-subscribe systems and also provide a translation tool to automatically generate analysis models. Although the ideas presented in these papers are similar to those described here, there are some differences. We extend this work by adding the probabilistic environment and we also allow components to change their subscriptions at run-time. These proposals are also extended in [9], which improves the representation of events, event delivery policies, and event-method bindings. Finally, [10] presents a transformation framework for the approach.

In [11], the authors present a compositional reasoning framework for verifying publish-subscribe systems. A system specification is decomposed into the properties of its components, which are then model checked separately under several environmental assumptions. The main drawback of this approach is the difficulty in decomposing the specification and providing appropriate assumptions.

3 Abstract Publish-Subscribe Architecture

The common denominator of the many variants proposed for the publish-subscribe paradigm is the decoupling in time and space of application components [1–3].

Fig. 1 illustrates the abstract model of a publish-subscribe infrastructure that we use for analysis via model checking. It comprises a dispatcher, the components, and a number of buffers that describe the transmission channels. Each buffer buf is characterized by a maximum size, MAX_buf; n_buf denotes the current number of elements stored in buffer buf. All buffers adopt a `FIFO` policy.

Components send their messages (i.e., publications, subscriptions, and unsubscriptions) to the dispatcher, by inserting them into buffer *bpi*. The message is tagged with the component that delivered the message. Messages are eventually transferred into buffer *bpo*, which models the input buffer to the dispatcher. Moving a message from *bpi* to *bpo* reflects the physical operation of message transmission. Since the message is tagged with the name of the originating component[3], it is possible to associate different probabilities with the delivery of messages from different components to the dispatcher. The messages delivered from the dispatcher are inserted into buffer *bni*. Each component Ci has an input buffer $bno[i]$ in which it receives the notifications. The transfer of a message from *bni* to a buffer $bno[i]$ reflects the physical operation of message transmission from the dispatcher to the notified component. The messages are transferred from *bni* into all the buffers of the target components to which it has to be delivered, based on the routing information stored in the subscription table. Again, a probability can be associated with the transfer of a message from *bni* to its target buffer in order to model unreliable channels.

We make the following assumptions, under which systems are analyzed via model checking: (a) transmission channels preserve the order of messages deliv-

[3] Tags are then removed when the messages are dispatched to their subscribers.

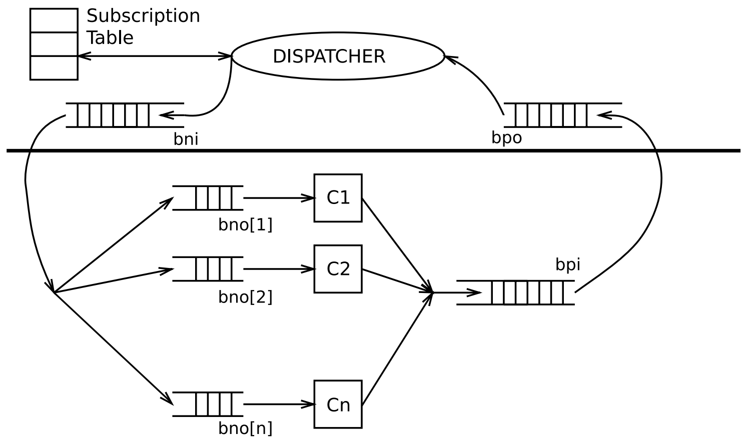

Fig. 1. Publish-subscribe infrastructure

ered through them, (b) messages are dropped in case of buffer overflow, and (c) transmission channels are unreliable. Messages may be lost during transmission according to a certain probability, which varies from component to component. We denote by p_i the probability of message loss for the i-th component[4]. We also introduce some timed parameters that are relevant to represent time-related properties of the system: TPL and TPH are the minimal and maximal time delay for the dispatcher to process a message, TDL and TDH are the minimal and maximal time delay for the dispatcher to process a subscription/unsubscription, and TRL and TRH are the minimal and maximal transmission delays of a message through the communication channels.

The thorough analysis of these parameters might lead to the explosion of the state space. If we consider all the combinations among the possible values for these parameters, we would easily obtain an unmanageable finite state model. Some *domain-specific* abstractions can help us constrain the problem and obtain a model more suitable for analysis. For example, if we consider an embedded system whose components are distributed over a local area network, we can easily assume that $TRL = TRH = 0$ since transmission delay can be ignored, and thus the number of possible different states (combination of parameter values) decreases.

[4] Notice that, since we model subscriptions and messages delivery with different buffers, our model allows to assign different probabilities to each component depending on the direction of the communication.

4 Probabilistic Timed Automata

Probabilistic timed automata provide a modeling framework for time-related systems with probability. Our definition is derived from timed automata [12,13], and we also follow the definitions in [14–16] with minor modifications.

A finite **discrete probability distribution** over a set S is a mapping $p : S \rightarrow [0,1]$ such that $\sum_{s \in S} p(s) = 1$ and the set $\{s | s \in S \text{ and } p(s) > 0\}$ is finite. The set of all finite discrete probability distributions over S is denoted by $\mu(S)$.

A **Markov decision process** is a discrete time stochastic process characterized by a set of states; in each state there are several actions from which the decision maker can choose. For a state s and an action a, a state transition function $p(s')$ determines the transition probabilities to the next state s'. A Markov decision process can be represented as a tuple $(Q, Steps)$, where Q is a set of states, and $Steps : Q \rightarrow 2^{\mu(Q)}$ is a function assigning a set of probability distributions to each state.

A **clock** is a real-valued variable which increases at a given rate. Let X be a set of clock variables, ranging over the nonnegative real numbers R^+. A valuation of X assigns a nonnegative real value to every clock in set X. We denote the set of all clock valuations of X with R^X.

A clock constraint is an inequality of the form $x \sim c$ or $x_i - x_j \sim c$, where $\sim$ is an operator in $\{<, \leq, >, \geq\}$ and c is a nonnegative integer number or infinity. A **clock zone** is a convex subset of the valuation space R^X described by a conjunction of constraints. Let $Z(X)$ be the set of all zones of X.

Given a clock zone $\lambda \in Z(X)$ and a valuation $v \in R^X$, $\lambda(v)$ is the boolean value obtained by replacing each occurrence of a clock $x \in X$ with $v(x)$. If $\lambda(v) = true$, we say that v satisfies λ, denoted as $v \rhd \lambda$.

Definition 1. *Let AP be a fixed, finite set of atomic propositions. A* probabilistic timed automaton *is a 7-tuple* $M = (S, S_0, L, X, inv, prob, \langle\tau_s\rangle_{s \in S})$*, where*

– S *is a finite set of* locations.
– s_0 *is the initial location.*
– $L : S \rightarrow 2^{AP}$ *is a labeling function that associates each location* $s \in S$ *with the set* $L(s)$ *of atomic propositions that are valid in* s*.*
– X *is a finite set of clocks.*
– $inv : S \rightarrow Z(X)$ *is a mapping that associates each location with an invariant condition.*
– $prob : S \rightarrow P_{fn}(\mu(S \times 2^X))$ *is a mapping function that associates each location with a finite and non-empty set of discrete probability distributions on* $S \times 2^X$*.*
– $\langle\tau_s\rangle_{s \in S}$ *is a family of functions where for any* $s \in S$*,* $\tau_s : prob(s) \rightarrow Z(X)$ *associates each* $p \in prob(s)$ *with an enabling condition.*

A state of a probabilistic timed automaton is a pair $\langle s, v\rangle$, where $s \in S$, $v \in R^X$, and $v \rhd inv(s)$. The system starts in location s_0, with all clocks initialized to 0. The values of all the clocks increase uniformly as time passes.

If we assume that the present state is $\langle s, v\rangle$, a probabilistic timed automaton has two basic types of transitions:

– *Delay transition*: the system can remain in the current location s and lets time pass, provided that the invariant condition in s can continuously be satisfied while time passes.
– *Action transition*: the system can make a discrete transition according to any probability distribution in $Prob(s)$ whose enabling condition is satisfied by the current time valuation v.

These concepts are exemplified by the automaton of Fig. 2. It consists of: two locations s_0 and s_1, two clocks x_0 and x_1, and two probabilistic distributions, a and b, associated with s_0 and s_1, respectively. The first distribution (a) defines a discrete transition from s_0 to s_1 with probability 1. The second distribution (b) defines a discrete transition from s_1 to s_0 again with probability 1. s_0 is the initial location, and the automaton starts with state $(s_0, x_0 = 0, x_1 = 0)$. Before the clock x_0 reaches the value 4, the automaton may remain in the location s_0 (delay transition). After 2 time units, the enabling condition for the discrete transition from s_0 to s_1 is satisfied, and the automaton can either move to the location s_1 (action transition) or remain in location s_0 (delay transition).

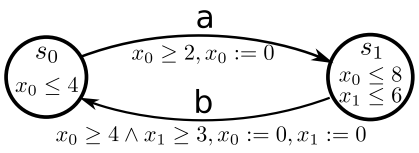

Fig. 2. An example of probabilistic timed automaton

5 Implementation

In this section, we explain how publish-subscribe systems can be formally specified and verified by means of probabilistic timed automata.

As discussed earlier, a model for a publish-subscribe system can be separated into two parts: the reusable part, consisting of the dispatcher and the transmission channels, and the problem-specific components. Following the model illustrated in Fig. 1, we start the presentation with the probabilistic timed automata that model the dispatcher and transmission channels, respectively, and then we continue with the application-specific components.

5.1 Dispatcher

Fig. 3 shows the probabilistic timed automaton that models the dispatcher. The automaton starts in location 0. The enabling conditions and actions are attached to the transitions as follows:

$$cond_1 \wedge cond_2 \wedge \cdots \wedge cond_m, act_1, act_2, \ldots, act_n$$

The semantics of labels is that if the conjunction of the conditions in the first part of the label holds, the sequence of actions that follows is performed atomically before the transition terminates.

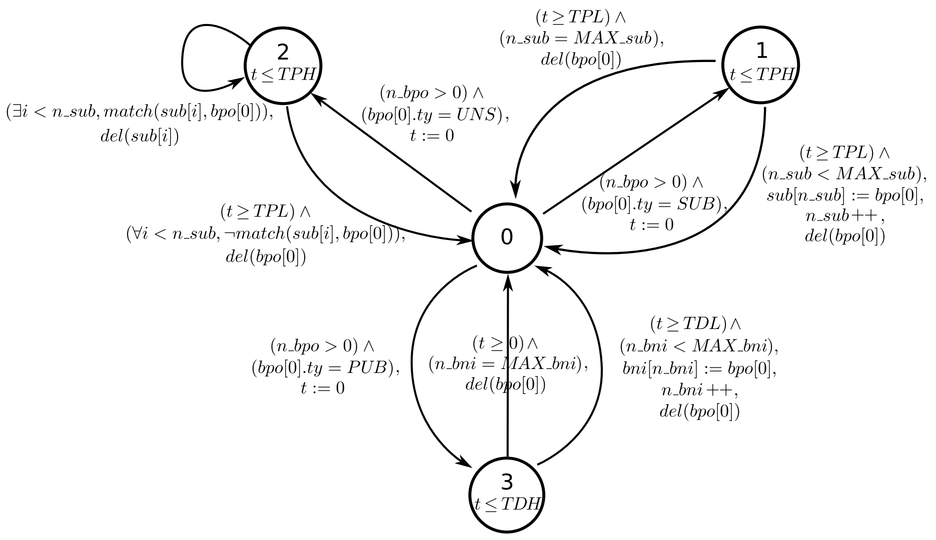

Fig. 3. Probabilistic timed automaton modeling the dispatcher

The dispatcher monitors the receiving buffer *bpo*. If it is not empty, the dispatcher fetches the first message in the buffer. If the message is a subscription, the dispatcher moves to location 1 by resetting timer t. The invariant in location 1 means that the dispatcher may process the message for at most TPH time units. Before t reaches the timeout, the dispatcher leaves location 1 and goes back to location 0 by either recording this subscription in the table (if the table is not full), or by dropping this subscription (if the table is full).

If the message is an unsubscription, the dispatcher moves to location 2 by resetting timer t. Then the dispatcher searches all the entries in the subscription table and removes those that match this unsubscription. Finally, it returns back to location 0 after at least a time delay. If the message is a publication, the dispatcher moves to location 3 by resetting timer t. Note that if the buffer *bni* is full, the dispatcher does not transfer the message and moves back to location 0. Otherwise the dispatcher transfers the message to buffer *bni* after a time delay. Operation `match`, which labels some of the transitions, performs the matching of a message against a subscription, and evaluates to true if its arguments match. The function encapsulates the details of the specific linguistic mechanisms supported by the publish-subscribe middleware to specify the matching. Operation `del` removes an element from a buffer (or subscription table).

5.2 Channels

The probabilistic timed automaton of Fig. 4 models the transmission of messages from components to the dispatcher. It commences in location 0. If buffer *bpi* is not empty, the automaton moves to location 1 by resetting timer t. The transitions exiting location 1 describe the fact that the channel may drop the message if the receiving buffer is full, or perform the probabilistic transmission otherwise. The probabilistic transition is drawn in the Fig. 4 as two directed shared edges connected by an arc. For readability reasons, we only show one example of a probabilistic transition, although there should be one for each component. The enabling condition is attached to the arc, and the actions are attached to the two edges. The message is lost with probability $p[i]$, and the message arrives at the receiving buffer with probability $1 - p[i]$. Notice that the message loss probability $p[i]$ can be different for different components. Moreover, this model does not distinguish if transmitted messages are subscriptions, unsubscriptions, or notifications; the channel only takes care of queueing the messages in the buffer and then the dispatcher deals with them.

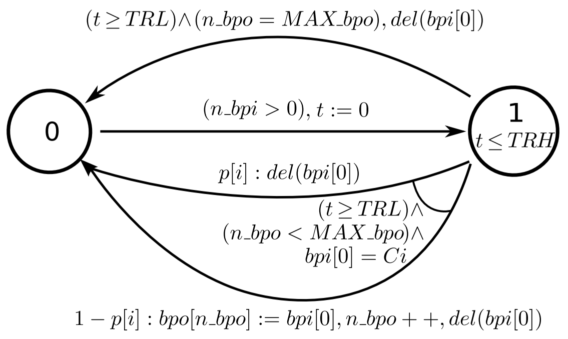
$(t \geq TRL) \wedge (n\_bpo = MAX\_bpo), del(bpi[0])$
0
$(n\_bpi > 0), t := 0$
1
$t \leq TRH$
$p[i] : del(bpi[0])$
$(t \geq TRL) \wedge$
$(n\_bpo < MAX\_bpo) \wedge$
$bpi[0] = Ci$
$1 - p[i] : bpo[n\_bpo] := bpi[0], n\_bpo++, del(bpi[0])$

Fig. 4. Probabilistic timed automaton modeling message transmission to the dispatcher

The probabilistic timed automaton of Fig. 5 models the notifications, i.e., the delivery of messages to the registered components. It starts in location 0 and if buffer *bni* is not empty, the channel moves to location 1 by initializing variable j to 0. The sequence of transitions from location 1 to locations 2 and 3, to end in location 1, models the notification of a message to a component. The transition from location 1 to location 2 models the match of the message with the j-th subscription. Variable $tmp1$ is used to record the identifier of the component that delivered the subscription, and variable $tmp2$ is used to record the number of elements currently stored in buffer $bno[tmp1]$. The self-loop transition in location 1 describes the mismatch of the j-th subscription with the message; the value of j is also incremented by 1.

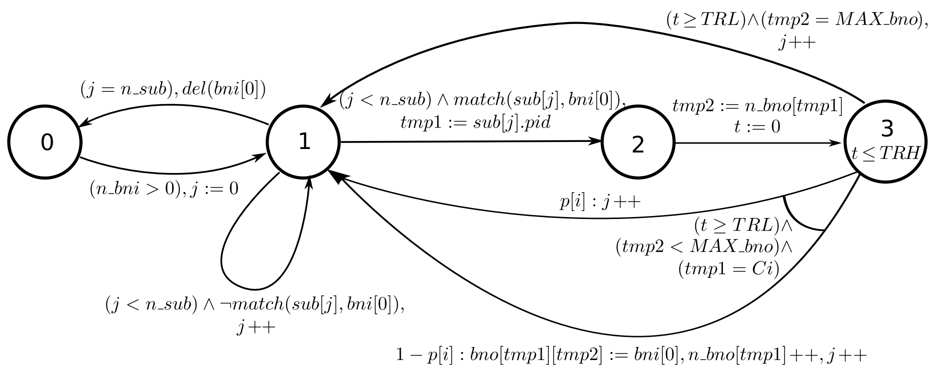
$(t \geq TRL) \wedge (tmp2 = MAX\_bno)$,
$j{+}{+}$
$(j = n\_sub), del(bni[0])$
$(j < n\_sub) \wedge match(sub[j], bni[0])$,
$tmp1 := sub[j].pid$
$tmp2 := n\_bno[tmp1]$
$t := 0$
0
1
2
3
$t \leq TRH$
$(n\_bni > 0), j := 0$
$p[i] : j{+}{+}$
$(t \geq TRL) \wedge$
$(tmp2 < MAX\_bno) \wedge$
$(tmp1 = Ci)$
$(j < n\_sub) \wedge \neg match(sub[j], bni[0])$,
$j{+}{+}$
$1 - p[i] : bno[tmp1][tmp2] := bni[0], n\_bno[tmp1]{+}{+}, j{+}{+}$

Fig. 5. Probabilistic timed automaton modeling notification

5.3 Application components

This section describes how to model application-specific components. The infrastructure provides the following operations to the designers of application components to let them communicate:

- `subscribe(pid,t)`: component *pid* subscribes to messages that match pattern t.
- `unsubscribe(pid,t)`: component *pid* withdraws its former subscriptions that match pattern t.
- `publish(k)`: a component publishes message k.
- `consume(k)`: a component removes message k from its receiving buffer *bno*.

As defined in Section 3, we assume that operation `consume` takes at least TCL and at most TCH time units. Probabilistic timed automata allow us to model the above operations as shown in Fig. 6. Component designers specify the behavior of components by means of statechart diagrams, and use the above operations to model the interaction among components. They can then translate the statechart diagrams to probabilistic timed automata according to the translation rules defined in Fig. 6.

6 Example application

This section applies our approach to a simple example taken in the domain of embedded control systems. The example consists of two sensors, a main processor, and an actuator. The main processor reads the responses from the sensors, and then feeds the actuator. This system can be implemented by using a publish-subscribe architecture as shown in Fig. 7.

First, the main processor publishes events to request data from the sensors. Then, the main processor, which receives responses from the sensors, publishes a

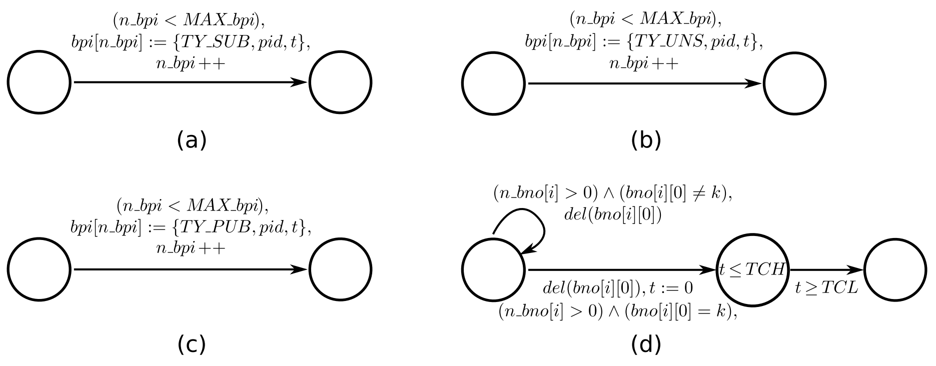
(n_bpi < MAX_bpi),
bpi[n_bpi] := {TY_SUB, pid, t},
n_bpi++
(a)
(n_bpi < MAX_bpi),
bpi[n_bpi] := {TY_UNS, pid, t},
n_bpi++
(b)
(n_bpi < MAX_bpi),
bpi[n_bpi] := {TY_PUB, pid, t},
n_bpi++
(c)
(n_bno[i] > 0) ∧ (bno[i][0] ≠ k),
del(bno[i][0])
t ≤ TCH
t ≥ TCL
del(bno[i][0]), t := 0
(n_bno[i] > 0) ∧ (bno[i][0] = k),
(d)

Fig. 6. Probabilistic timed automata modeling communication operations: (a) `subscribe(pid,t)`, (b) `unsubscribe(pid,t)`, (c) `publish(k)`, and (d) `consume(k)`

message to feed the actuator[5]. The sensors take from $TSDL$ to $TSDH$ time units to give their responses, and the actuator uses from $TADL$ to $TADH$ time units to be fed. Since the message may be lost during the transmission, we assume an upper bound TMW for the main processor to wait for data. We start a timer as soon as the main process begins to work. If a time-out occurs, the main process terminates and reports a failure.

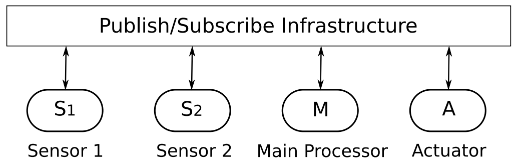

Fig. 7. Publish-subscribe architecture of the embedded control system

Fig. 8 and Fig. 9 show the statechart diagrams for the sensors and the main processor, respectively. The statechart diagram for the actuator is not illustrated here since it is similar to the one for the sensors. The corresponding probabilistic timed automata can be obtained from the statechart diagrams by applying the translation rules of Fig. 6. For example, the probabilistic timed automaton obtained from the statechart diagram for sensors is shown in Fig. 10. The probabilistic timed automaton of the main processor can be obtained similarly. In this example, subscriptions cannot change dynamically (i.e., while the system executes); they are defined statically. Hence the complete model contains the above described components and the dispatcher presented in Section 5, where

[5] Notice that all the messages are mediated through the dispatcher.

subscription and unsubscription messages are ignored. Accordingly, the model in Fig. 4 only handles publication messages.

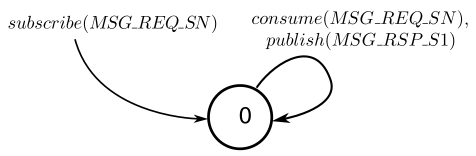

Fig. 8. Statechart diagram for a sensor

The choice of only considering static subscriptions is not due to limitations of the proposed approach, but to the nature of the example application, since embedded systems are often statically configured. However, should we need to add dynamic subscriptions, the components ought to be modified by adding two new outgoing transitions from location 0 to model subscription and unsubscription requests, respectively. These transitions are taken with a given probability to simulate the success of these operations and thus how the scenario changes[6]. If one of these transitions is taken, the automaton moves to a new location, from which we model the subscription (or unsubscription) request.

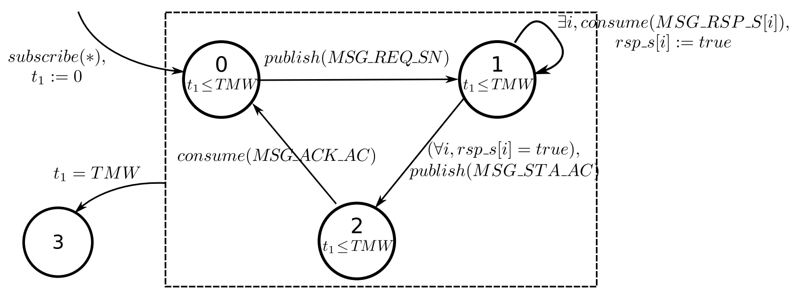

Fig. 9. Statechart diagram for the main processor

6.1 PRISM model

PRISM [17–19] is the model checker we selected to verify our models. PRISM is a probabilistic model checker developed at the University of Birmingham and is a tool for the design and analysis of systems that exhibit probabilistic behaviors.

[6] Notice that in this example, this probability is 0, since with do not permit dynamic subscriptions and unsubscriptions and thus the arcs are skipped.

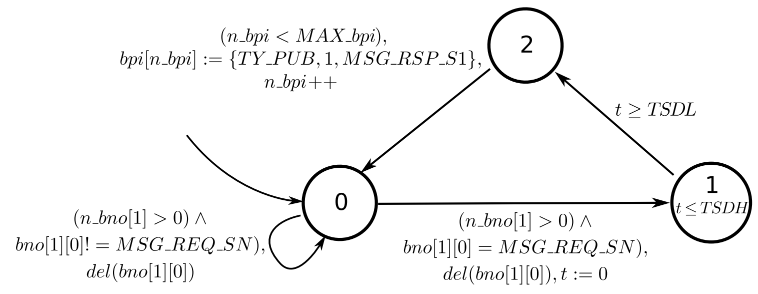

Fig. 10. Probabilistic timed automata for a sensor

It supports three types of probabilistic models: Discrete-Time Markov Chains (DTMCs), Markov Decision Process (MDPs), and Continuous-Time Markov Chains (CTMCs). Models are specified in a simple, high-level modeling language, which is a variant of the Reactive Modules formalism of Alur and Henzinger [20]. Properties are described by the PRISM property specification language, which is based on the two probabilistic temporal logics called Probabilistic Computation Tree Logic (PCTL) [21,22] and Continuous Stochastic Logic (CSL) [23,24].

Since we are interested in modeling both probabilistic (unreliable channels with message loss) and non-deterministic (time delays) behaviors of publish-subscribe systems, we decided to adopt the MDP formalism, which allows us to mix the two different types of behaviors. The translation from probabilistic timed automata to PRISM models is easy. Each automaton in the publish-subscribe system corresponds to a PRISM module. The buffers and the subscription table are rendered as global variables. Since all the timers should run at the same rate, it is required that all the time passing actions be synchronized. For example, the dispatcher staying in location 1 with time progressing is described in PRISM as:

```
[time] s_dp=1 & t_dp<TPH -> t_dp'=min(t_dp+1,TPH);
```

Similarly, sensor 1 staying in location 2 with time progressing can be described in PRISM as:

```
[time] s_s1=1 & t_s1<TSDH -> t_s1'=min(t_s1+1,TSDH);
```

Notice that these two commands are labeled with the same action `time` to mean that these two commands need to be synchronized.

PRISM models can be augmented with *rewards structures*[7], which associate real values with certain states or transitions of the model. With reward structures, PRISM can be used to reason about the properties related to the expected values of these rewards. In our model, we assign a reward of 1 to all the transitions labeled with action *time*. All the others maintain 0 as default value. This way we can verify the properties related to expected time.

[7] Interested readers can refer to [25,26] for a detailed presentation.

6.2 Experimental Results

This section presents some results obtained by using the PRISM model checker to verify the example system.

Service provision The first concern is to understand if the task finishes successfully. This means that the automaton of the main processor must end in location 0 without generating a time-out. This can be expressed in PRISM as:

```
label "succeed" = s_mn=0 & t_mn>0;
Pmax=?[true U "succeed"], Pmin=?[true U "succeed"]
```

Notice that the probabilities for an MDP can only be computed after nondeterminism is resolved. Here we have two types of probabilities: P_{max} and P_{min} correspond to the resolutions that all the delays take the minimum or maximum values. We checked the effect of p_0 (message loss probability for the main processor) on the two probabilities and Fig. 11 shows the results. Notice that the two probabilities behave the same way, so we use the term *probability* in Fig. 11 to refer to both the probabilities. Similar results can be obtained for other message loss probabilities. It is easy to understand that probability P_{max} (or P_{min}) decreases as the message loss probability increases.

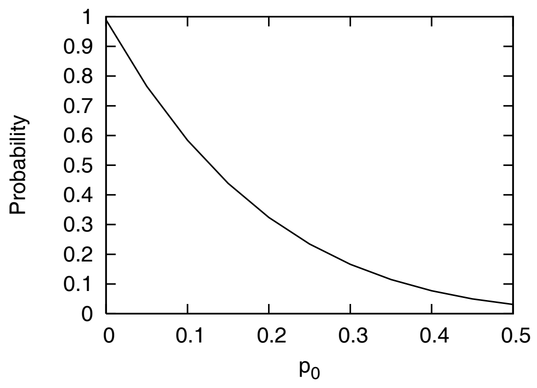

Fig. 11. Effect of p_0 on the probability (P_{max}, P_{min})

Average time Our second concern is about the average time the system takes to complete the task. It can be expressed in PRISM as:

```
label "terminate" = (s_mn=0 & t_mn>0) | (s_mn=3);
Rmax=?[F "terminate"], Rmin=?[F "terminate"]
```

where `R=? [F prop]` is a reward-based property. It accumulates rewards along each path until `prop` is satisfied, and then returns the expected value [26].

We have tested the effect of p_0 on both R_{max} and R_{min}. Since we assign a reward of 1 to all the transitions labeled with action *time*, the values of R_{max} and R_{min} just give the average times in the case of maximum and minimum delays, respectively. The results are plotted in Fig. 12 and show that when p_0 increases, the probability of the task to successfully finish decreases, as a result the average time increases accordingly. Needless to say, the system takes more time if the sub-tasks fail. There is also an upper bound for the two expected values.

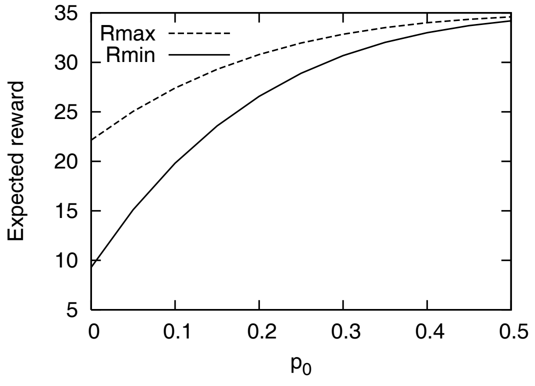

Fig. 12. The effect of TDL on $Rmin$

Effect of buffer size If we consider the probabilities computed in the first experiment, the values of P_{max} and P_{min} are the same. However, if we reduce the size of the corresponding buffers, the value of P_{min} may become 0. To understand this result, we need to consider the scenario in which all the sensors send their responses to the dispatcher at (almost) the same time. If the size of the receiving buffer is not large enough, or if the dispatcher cannot process existing messages quickly enough, there is a buffer overflow and the arrival of a new message would be dropped directly. The same situation can happen to the other buffers. Our experiments show that probabilistic model checking help optimize the appropriate size of buffers of the system model.

7 Conclusions and future work

This paper presented an approach to modeling and validating publish-subscribe systems by using probabilistic model checking. The infrastructure (transmission channels, dispatcher) is modeled by probabilistic timed automata. Application-specific components are modeled by statechart diagrams and then translated into probabilistic timed automata. The actual validation is carried out by using the PRISM model checker.

This paper presented some preliminary results we obtained so far, which are motivating us to keep investigating these ideas. Our long-term goal is to design an integrated tool-set for the multi-perspective validation of the highly dynamic software architectures, which are becoming increasingly important and widespread in practice. Publish-subscribe architecture are a first notable class of such architectures, on which we initially focused our research.

References

1. Carzaniga, A., Rosenblum, D.S., Wolf, A.L.: Design and evaluation of a wide-area event notification service. ACM Transactions on Computer Systems **19**(3) (2001) 332–383
2. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM Comput. Surv. **35**(2) (2003) 114–131
3. Cugola, G., Picco, G.P.: Reds: a reconfigurable dispatching system. In: SEM '06: Proceedings of the 6th international workshop on Software engineering and middleware, New York, NY, USA, ACM Press (2006) 9–16
4. Zanolin, L., Ghezzi, C., Baresi, L.: An approach to model and validate publish/subscribe architectures. In: Proceedings of the SAVCBS'03 Workshop, Helsinki, Finland (2003)
5. Baresi, L., Ghezzi, C., Mottola, L.: Towards fine-grained automated verification of publish-subscribe architectures. In Najm, E., Pradat-Peyre, J.F., Donzeau-Gouge, V., eds.: FORTE. Volume 4229 of Lecture Notes in Computer Science., Springer (2006) 131–135
6. Baresi, L., Ghezzi, C., Mottola, L.: On accurate automatic verification of publish-subscribe architectures. In: (To appear) Proceedings of the 29th International Conference on Software Engineering (ICSE07), Minneapolis (MN, USA) (2007)
7. Garlan, D., Khersonsky, S.: Model checking implicit-invocation systems. In: Proc. of the 10th Int'l Workshop on Software Specification and Design. (2000) 23–30
8. Garlan, D., Khersonsky, S., Kim, J.S.: Model checking publish-subscribe systems. In: Proc. of the 10th Int'l SPIN Workshop on Model Checking of Software. (2003)
9. Bradbury, J.S., Dingel, J.: Evaluating and improving the automatic analysis of implicit invocation systems. In: FSE. (2003)
10. Zhang, H., Bradbury, J.S., Cordy, J.R., Dingel, J.: A transformational framework for testing and model checking implicit invocation systems. In: Proc. Int. Work. on Distr. Event-Based Systems (DEBS'04). (2004)
11. Caporuscio, M., Inverardi, P., Pelliccione, P.: Compositional verification of middleware-based software architecture descriptions. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering, Washington, DC, USA, IEEE Computer Society (2004) 221–230
12. Alur, R., Courcoubetis, C., Dill, D.: Model-checking for real-time systems. In: Proceedings of Fifth Annual IEEE Symposium on Logic in Computer Science. (1990) 414–425
13. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. **126**(2) (1994) 183–235

14. Kwiatkowska, M., Norman, G., Segala, R., Sproston, J.: Automatic verification of real-time systems with discrete probability distributions. Theoretical Computer Science **282** (2002) 101–150
15. Kwiatkowska, M., Norman, G., Sproston, J., Wang, F.: Symbolic model checking for probabilistic timed automata. In Lakhnech, Y., Yovine, S., eds.: Proc. Joint Conference on Formal Modelling and Analysis of Timed Systems and Formal Techniques in Real-Time and Fault Tolerant Systems (FORMATS/FTRTFT'04). Volume 3253 of LNCS., Springer (2004) 293–308
16. Kwiatkowska, M., Norman, G., Parker, D., Sproston, J.: Performance analysis of probabilistic timed automata using digital clocks. Formal Methods in System Design **29** (2006) 33–78
17. Kwiatkowska, M., Norman, G., Parker, D.: Probabilistic symbolic model checking with PRISM: A hybrid approach. International Journal on Software Tools for Technology Transfer (STTT) **6**(2) (2004) 128–142
18. Kwiatkowska, M., Norman, G., Parker, D.: Quantitative analysis with the probabilistic model checker PRISM. Electronic Notes in Theoretical Computer Science **153**(2) (2005) 5–31
19. Hinton, A., Kwiatkowska, M., Norman, G., Parker, D.: PRISM: A tool for automatic verification of probabilistic systems. In Hermanns, H., Palsberg, J., eds.: Proc. 12th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'06). Volume 3920 of LNCS., Springer (2006) 441–444
20. Alur, R., Henzinger, T.A.: Reactive modules. Formal Methods in System Design: An International Journal **15**(1) (1999) 7–48
21. Hansson, H., Jonsson, B.: A logic for reasoning about time and reliability. Formal Aspects of Computing **6**(5) (1994) 512–535
22. Bianco, A., de Alfaro, L.: Model checking of probabilistic and nondeterministic systems. In Thiagarajan, P., ed.: FSTTCS: Foundations of Software Technology and Theoretical Computer Science. Volume 15 of volume 1026 of LNCS., Springer (1995) 499–513
23. Aziz, A., Sanwal, K., Singhal, V., Brayton, R.K.: Verifying continuous time markov chains. In Rajeev Alur, Thomas A. Henzinger, eds.: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification. Volume LNCS 1102., London, UK, Springer-Verlag (1996) 269–276
24. Baier, C., Katoen, J.P., Hermanns, H.: Approximate symbolic model checking of continuous-time markov chains. In: CONCUR '99: Proceedings of the 10th International Conference on Concurrency Theory, London, UK, Springer-Verlag (1999) 146–161
25. Kwiatkowska, M., Norman, G., Pacheco, A.: Model checking expected time and expected reward formulae with random time bounds. In: Proc. 2nd Euro-Japanese Workshop on Stochastic Risk Modelling for Finance, Insurance, Production and Reliability. (2002)
26. Prism user manual, http://www.cs.bham.ac.uk/ dxp/prism/manual/.