The following publication W. Zhu, J. Cao and M. Raynal, "Predicate Detection in Asynchronous Distributed Systems: A Probabilistic Approach," in IEEE Transactions on Computers, vol. 65, no. 1, pp. 173-186, 1 Jan. 2016 is available at https://doi.org/10.1109/TC.2015.2409839

# Predicate Detection in Asynchronous Distributed Systems: A Probabilistic Approach

Weiping Zhu; Jiannong Cao, *Fellow, IEEE;* and Michel Raynal

**Abstract**—In an asynchronous distributed system, a number of processes communicate with each other via message passing that has a finite but arbitrary long delay. There is no global clock in that system. Predicates, denoting the states of processes and their relations, are often used to specify the information of interest in such a system. Due to the lack of a global clock, the temporal relations between the states at different processes cannot be uniquely determined, but have multiple possible circumstances. Existing works of predicate detection are based on the *definitely* modality or the *possibly* modality, denoting that a predicate holds in all of the possible circumstances or in one of them, respectively. No information is provided about the probability that a predicate will hold, which hinders the taking of countermeasures for different situations. Moreover, the detection is based on single occurrence of a predicate, so the results are heavily affected by environmental noise and detection errors. In this paper, we propose a new approach to predicate detection to address these two issues. We generalize the *definitely* and *possibly* modalities to an occurrence probability to provide more detailed information, and further investigate how to detect multiple occurrences of a predicate. We propose a unified algorithm framework for detecting various types of predicates and demonstrate the use of it for three typical types of predicates, including simple predicates, simple sequences, and interval-constrained sequences. Theoretical proofs and simulation results show that our approach is effective and outperforms existing approaches.

**Index Terms**—Predicate detection, asynchronous distributed systems, occurrence probability, occurrence times.

## 1 Introduction

In an asynchronous distributed system, a collection of distributed computation entities called "processes" communicate with each other via message passing. The message passing has a finite but arbitrary long delay. No global clock exists in that system. The communication requirement in such a system is quite relaxed, and hence suitable for a wide range of applications, including network management [1], [2], [3], circuit design [4], [5], robot control [6], [7], [8], and pervasive computing [9], [10]. Predicates, denoting the states of processes and their relations, are often used to specify the information of interest in these applications. For example, a potential explosion alarm of natural gas based on two sensors can be specified by a predicate: *"the value of the temperate sensor is greater than* $580^o c$ *and at the same time the value of the concentration sensor is between 4.4% and 17% (by volume of air)."*

Detecting predicates in asynchronous distributed systems is quite challenging. Due to the lack of a global clock, the sequence of global states that a system has passed in a execution of a program, called an observation, has different possibilities. A predicate may hold in some of the possible observations, but may not in others.

In existing works, the *definitely* and *possibly* modalities are used to enable solutions to detect predicates [11], [12]. It is claimed that a predicate definitely holds if it can be detected in all observations, and possibly holds if it can be detected in at least one observation. These modalities work well in their original target applications, program testing and debugging, but are insufficient for many other applications, such as environmental monitoring and network analysis. There are two issues that need to be addressed: First, the *definitely* modality is too strict and the *possibly* modality provides limited information. It is highly desirable to determine the probability that a predicate will hold. We call this the *occurrence probability*. Second, considering that environmental noise and detection errors exist frequently, a more reasonable practice is to take into account the multiple occurrence of a predicate rather than a single occurrence. For each occurrence of the predicate, we can specify its occurrence probability. This requirement exists in many applications. For example, in a smart home [13], in order to infer that a person is reading and then adjust the light at home, it is necessary to detect that the person turns the pages of a book for a number of times with a proper occurrence probability. Another example can be found in social networks [14], where the friendship between two persons can be inferred by using the number of times that they meet, and each time they meet is detected with a high level of

- *Weiping Zhu is with the International School of Software, Wuhan University, China and Department of Computing, The Hong Kong Polytechnic University, Hong Kong E-mail: cswpzhu@gmail.com.*
- *Jiannong Cao is with the Department of Computing, The Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong. E-mail: csjcao@comp.polyu.edu.hk.*
- *Michel Raynal is with the Institut Universitaire de France & IRISA-INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France. E-mail: raynal@irisa.fr.*

probability.

In this paper, we propose a new predicate detection approach to address the aforementioned two issues. We first extend *definitely* and *possibly* modalities to a more generic occurrence probability. The *definitely* modality corresponds to an occurrence probability of 1, and the *possibly* modality corresponds to occurrence probabilities of between 0 and 1. We further introduce the concept of the occurrence times of a predicate subject to an occurrence probability for each single detection. We propose a unified algorithm framework to ease the detection of various kinds of predicates, considering the occurrence probability and occurrence times. Three typical predicates, including simple predicates, simple sequences, and interval-constrained sequences, are used to demonstrate the use of our algorithm framework. Theoretical proofs and extensive simulations are conducted to validate the effectiveness of the proposed algorithms. In summary, this paper makes the following contributions:

- We generalized the *definitely* and *possibly* modalities to an occurrence probability, providing more detailed information to the user.
- We proposed the concept of the occurrence times of a predicate subject to an occurrence probability, which is more reliable than focusing on the single occurrence of a predicate.
- We designed a unified algorithm framework for the detection of various kinds of predicates, based on the occurrence probability and occurrence times.
- We proposed detailed detection algorithms for simple predicates, simple sequences, and interval-constrained sequences. The evaluation results show that the proposed algorithms work effectively and outperform existing approaches.

The rest of the paper is organized as follows: Section 2 describes the preliminary work. We then compose a research framework for predicate detection in section 3 and formulate our problem in section 4. Our algorithm is proposed in section 5 and further discussed in section 6. Simulation results are reported in section 7. Related works are summarized in section 8. We conclude the paper in section 9.

## 2 PRELIMINARY

We first present some preliminary work on predicate detection in asynchronous distributed systems.

### 2.1 Asynchronous Distributed Systems

An asynchronous distributed system includes a number of $n$ processes $P_i$ $(i = 1...n)$ and an underlying communication network. The communication network facilitates the exchanging of messages among processes, which involves a finite but arbitrary delay. During the execution of a distributed program

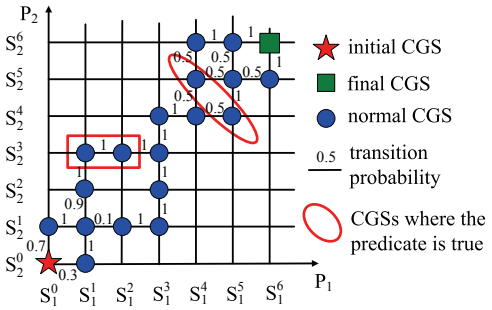


Fig. 1: An example of a lattice

in such a system, each process $P_i$ records its local computation, which includes alternate local states and events $s_i^0, e_i^0, s_i^1, e_i^1, ..., e_i^j, s_i^j$ [15], where $s_i^k$ denotes the $k$th local state and $e_i^k$ denotes the $k$th event, at the process $i$. A state can be described by the values of the variables in the process (e.g., registers, stacks, memory content, etc.). The $k$th event triggers the transition from the *(k-1)*th state to the $k$th state. The events can be local events (i.e., changing the values of variables in the process) or external events (i.e., sending messages to or receiving messages from other processes).

A happen-before relation and a concurrent relation [16] exist among the states.

A state $s_a$ is said to happen before another state $s_b$, denoted by $s_a \rightarrow s_b$, if

1) $s_a$ is a state before $s_b$ in the same process.

2) the event just following state $s_a$ sends a message and the event just before state $s_b$ receives that message.

3) there is a state $s_c$ such that $s_a \rightarrow s_c$ and $s_c \rightarrow s_b$ [16].

If $s_a$ does not happen before $s_b$ and $s_b$ does not happen before $s_a$, we say that $s_a$ is concurrent with $s_b$, denoted by $s_a \parallel s_b$.

### 2.2 Global State, Observation, and Lattice

A global state of an asynchronous distributed system is a set of local states of processes, with one state from each process. A global state $C$ is called a consistent global state (CGS) if the local states included in it are pairwise concurrent:

$C = (s_1, s_2, ..., s_n), \forall i, j : 1 \leq i \neq j \leq n :: s_i \| s_j$

It denotes a snapshot of an asynchronous distributed system where each local state does not happen before the others [17], [18].

Two consistent global states $C_1$ and $C_2$ satisfy the *precede* relation ($C_1 \prec C_2$) if they differ in only one local state, say $s_{c_1}$ and $s_{c_2}$, such that $s_{c_1}$ is immediately before $s_{c_2}$. The *lead to* relation ($\rightsquigarrow$) is the transitive closure of the *precede* relation ($\prec$).

A sequence of CGSs starting from the initial CGS (including all of the local initial states) and ending at the final CGS (including all of the local final states) is called an observation of the execution of the program.

It is difficult to determine the CGSs that really exist during the execution of a program. Instead, more

CGSs are claimed to possibly have occurred, which leads to multiple observations. Predicates may hold in some of the observations but may not in others. The *definitely* and *possibly* [11], [12] modalities denote cases in which a predicate holds in all of the observations or in at least one observation, respectively.

It is observed that CGSs and their *precede* relations form a lattice [19]. An example of this can be seen in Fig. 1. The nodes (the pair of local states, e.g., $(S_1^1, S_2^1)$) denote CGSs, and the edges between two nodes denote their *precede* relations. A path from the initial node to the final node corresponds to an observation of the execution of the program. A cut of a lattice is defined as a set of CGSs that are concurrent with each other and across all the paths. For example, $\{(S_1^2, S_2^3), (S_1^3, S_2^2)\}$ is a cut. A lattice is an effective tool for detecting predicates.

### 2.3 Predicate

In distributed systems, a predicate is a boolean function defined on the states of processes [20], [21]. Predicates vary in stability and form.

A predicate can be a stable predicate or an unstable predicate [22]. A stable predicate is a predicate that remains true once it is true. An unstable predicate is a predicate that is true intermittently. The detection of unstable predicates is more generic and challenging than the detection of stable predicates, and hence is the focus of this paper.

Predicates take different forms. In [23], three typical forms are investigated, including the *simple predicate*, *simple sequence*, and *interval-constrained sequence*. In this paper, these three types of predicates are used to demonstrate the effectiveness of our approach. We briefly describe their definitions [23] as follows:

A simple predicate is a predicate defined on a single CGS. An observation $O$ satisfies a simple predicate $e$, if and only if there exists a CGS $C_i \in O$ such that $e$ is true, denoted by $C_i(e)$. Simple predicates can be further classified into *simple conjunctive predicates* and *simple relational predicates*. The former denotes a conjunctive expression of local states of processes, and the latter can specify arbitrary relations between them.

A simple sequence is a predicate defined on a sequence of CGSs. It is in the form of $e_1; e_2; ....e_m$, where $e_i$ $(1 \leq i \leq m)$ is a simple predicate specifying a desirable system state. An observation $O$ satisfies this simple sequence if and only if there exist $m$ distinct CGSs $C_1, C_2...C_m \in O$, such that

1) $C_1 \rightsquigarrow C_2 \rightsquigarrow ... \rightsquigarrow C_m$
2) $C_1(e_1) \wedge C_2(e_2) \wedge ... \wedge C_m(e_m)$.

An interval-constrained sequence extends a simple sequence to support undesirable states. It is in the form of $[\theta_1]e_1; ...[\theta_m]e_m; [\theta_{m+1}]$, where $e_i(1 \leq i \leq m)$ and $\theta_i(1 \leq i \leq m+1)$ are simple predicates, and $[\theta_i](1 \leq i \leq m+1)$ denotes that $\theta_i$ is false. An observation $O$ satisfies this predicate if and only if:

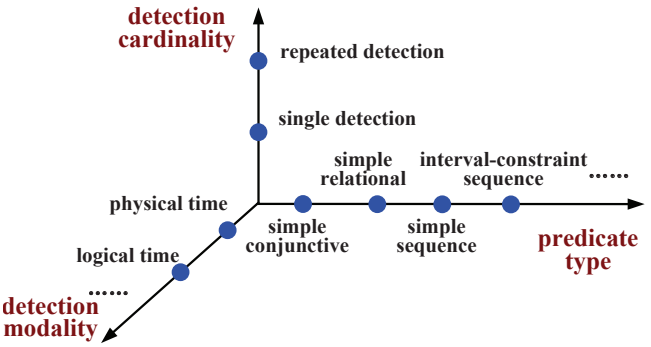


Fig. 2: Research framework for predicate detection

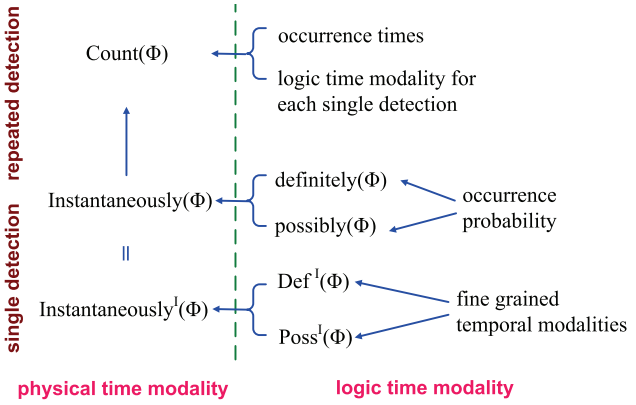


Fig. 3: Predicate detection under different modalities

1) when $m = 0$, $\forall C_i \in O :: \neg C_i(\theta_1)$
2) when $m \neq 0$, there exist $m$ distinct CGSs $C_1, C_2...C_m \in O$ such that
   a) $C_1 \rightsquigarrow C_2 \rightsquigarrow ... \rightsquigarrow C_m$
   b) $C_1(e_1) \wedge C_2(e_2) \wedge ... \wedge C_m(e_m)$
   c) $\forall C_k : C_k \rightsquigarrow C_1 :: \neg C_k(\theta_1)$
   d) $\forall C_k : C_m \rightsquigarrow C_k :: \neg C_k(\theta_{m+1})$
   e) $j = 2....m$, $\forall C_k : C_j \rightsquigarrow C_k \rightsquigarrow C_{j+1} :: \neg C_k(\theta_j)$

## 3 Research Framework for Predicate Detection

In this section, we compose a research framework for predicate detection, and illustrate the position of our work in this research field.

We categorize predicate detections according to three metrics: predicate type, detection cardinality, and detection modality, as shown in Fig. 2. Predicate type denotes the inherent properties of a predicate. It may refer to the forms of predicates, including *simple conjunctive*, *simple relational*, *simple sequence*, *interval-constraint sequence* [23], and others. More detailed classifications may also take the stability of predicates into consideration. Detection cardinality includes *single detection* and *repeated detection* [24]. The former is to validate whether a predicate is true at least one time, and the latter is to detect the predicate multiple times. Finally, detection modality includes *physical time modality* and *logical time modality*. They

denote the conditions to be satisfied for a predicate in terms of physical time and logical time, respectively.

Fig. 3 shows the classification of detection modality. The temporal relations specified by the predicates can be defined based on physical time or logical time. For physical time, it is assumed that all processes have a global clock. *Instantaneously* is the most widely used modality under physical time, denoting that processes are in certain states at the same time. This modality can be further classified into *Instantaneously*($\Phi$) and *Instantaneously*$^I(\Phi)$, depending on time points or time intervals used for the detection of predicate $\Phi$, respectively. *Instantaneously*($\Phi$) denotes that there exists a time point such that $\Phi$ is satisfied. *Instantaneously*$^I(\Phi)$ denotes that there exists a set of pairwise overlapped time intervals $I$ such that $\Phi$ is satisfied. *Instantaneously*($\Phi$) is equivalent to *Instantaneously*$^I(\Phi)$ [10], [25].

In distributed systems, physical time usually cannot work due to the lack of a global clock. Predicate detection is more often based on logical time. Existing work on that can be classified into two categories. The first category is based on the time points, using *definitely*($\Phi$) and *possibly*($\Phi$) [11], [12]. *definitely*($\Phi$) means that $\Phi$ is true in all observations, while *possibly*($\Phi$) means that $\Phi$ is true in at least one observation. The other category is based on time intervals, using $Def^I(\Phi)$ and $Poss^I(\Phi)$ [9]. $Def^I(\Phi)$ means that according to the analysis in logical time, a set of time intervals $I$ can be inferred to be pairwise overlapped in the physical time during which $\Phi$ is true. $Poss^I(\Phi)$ means that the time interval set $I$ may be pairwise overlapped, but cannot be validated. $Def^I(\Phi)$ and $Poss^I(\Phi)$ can be further investigated under the fine-grained relations of time intervals [25]. For *definitely*($\Phi$) and *possibly*($\Phi$), currently no further information is provided, especially on how probable it is that a predicate holds.

Different with single detection, repeated detection [24] is about determining how many times a predicate $\Phi$ holds, denoted by $Count(\Phi)$. The concerns here are two-fold: one is the detection in each occurrence, which is the same as single detection. The other is the times that a predicate holds under different criterions. Most current research focuses on single detection, and cannot be directly applied to repeated detection.

In this paper, we aim to fill some gaps in the research, focusing on providing more detailed information for *definitely* and *possibly* modalities, and investigating them in the context of repeated detection.

## 4 System Model and The Problem

We extend a lattice to support the transition probability between two consecutive CGSs. Based on that, we formulate the problem discussed in this paper.

### 4.1 Extended Lattice and the Occurrence Probability of Predicates

In existing works, a lattice describes only CGSs and the *precede* relations between them. In order to more precisely describe the transitions of CGSs, we augment the transition probability between two CGSs to corresponding edges of the lattice. Generally speaking, a CGS $C_i$ may have multiple following CGSs $C_{i1}, C_{i2}....C_{im}$. The transition probability from $C_i$ to $C_{ij}$ $(1 \leq j \leq m)$ can be arbitrary but satisfy $\sum_{1 \leq j \leq m} C_{ij} = 1$.

The specific values of transition probabilities can be determined through different approaches, including data mining and analyzing context information. For example, in an intelligent traffic system, suppose that there are two RFID readers $R_1$ and $R_2$ monitoring the number of vehicles around two neighboring intersections, one reader for each intersection. The values of both $R_1$ and $R_2$ are changed from 50 to 60. Using "($R_1$'s value, $R_2$'s value)" to denote a CGS, the initial CGS is (50, 50) and its following CGSs are (50, 60) and (60, 50). By analyzing historical traffic data, we may find that 80% of cases with the same initial CGS changed to (50, 60). We then set the transition probability from (50, 50) to (50, 60) as 0.8 and that from (50, 50) to (60, 50) as 0.2. One more example concerns the data aggregation of sensor networks [26]. In the case of a pair of nodes, one node sends the data to the other node. A probabilistic algorithm is used to determine which node is the sender and which is the receiver. The context information of the probabilistic algorithm can be used to determine the transition probabilities in this case. If information on constraints is lacking, it can be assumed that all following CGSs have an equal probability of being reached. Fig. 1 shows an example of an extended lattice.

Given a path of a lattice, $\Omega = C_1, C_2,....C_m$, where the transition probability from $C_i$ to $C_{i+1}$ $(i = 1...m-1)$ is denoted by $p_{i,i+1}$, the transition probability of the path is defined by

$prob(\Omega) = p_{1,2} \times p_{2,3} \times .... \times p_{m-1,m}$

It denotes the probability that the observation corresponding to $\Omega$ happens.

Given a lattice and a predicate, and assuming that $\Gamma$ is a set of paths in each of which the predicate holds, the occurrence probability of this predicate is defined by $\sum_{\Omega \in \Gamma} prob(\Omega)$. In Fig. 1, if we select $(S_1^0, S_2^0)$ as the initial node and $(S_1^2, S_2^3)$ as the final node, the occurrence probability of the predicate is $0.9$.

In this way, the *definitely* and *possibly* modalities are generalized to an occurrence probability. The *definitely* modality is a special case in that the occurrence probability is equal to 1, and the *possibly* modality denotes that the occurrence probability is between 0 and 1. By quantifying the occurrence probability, we can provide more detailed information to the user.

### 4.2 The Occurrence Times of a Predicate Subject to an Occurrence Probability

In predicate detection, users are often interested in the repeated occurrences of a predicate. Informally, in a scenario where a predicate becomes true intermittently, we call each time that the predicate becomes true *an occurrence* and the number of times that it becomes true *occurrence times*. The occurrence times can be obtained determinatively in a single observation. But considering all observations, the occurrence times relate to the occurrence probability. We formulate the problem as follows:

Suppose there is a set of observations $\Phi = \{O_i | i = 1...n\}$. Each observation $O_i$ has its occurrence probability $prob(O_i)$ and occurrence times $num(O_i)$. Let $\Phi' \subseteq \Phi$. Then $\Phi'$'s occurrence probability is defined by $\sum_{o \in \Phi'} prob(O)$ and its occurrence times is defined by $\min_{o \in \Phi'} num(O)$. Our problem is to find a subset of $\Phi$ such that its occurrence times are maximized and its occurrence probability is no less than $p$. The result is called the *occurrence times of the predicate subject to occurrence probability* $p$.

Lattice is an effective tool for solving this problem. An observation $O_i$ corresponds to a path $\Omega$ in the lattice, and then $prob(O_i)$ is equal to $prob(\Omega)$. As shown in Fig. 1, if the occurrence probability is 0.9, two occurrences of the predicate can be determined, one from $(S_1^0, S_2^0)$ to $(S_1^3, S_2^3)$ and the other from $(S_1^3, S_2^4)$ to $(S_1^6, S_2^6)$.

In many applications, such as smart home systems and social relation analysis systems as mentioned in section 1, multiple occurrences of a predicate with a high occurrence probability (but less than 1) are regarded as a more reliable result than a single occurrence, even under the *definitely* modality. Therefore, we think that this problem is an important one.

### 4.3 Further Discussion

It is observed that the solution of the above problem can also be used to determine the occurrence probability of a predicate.

When calculating the occurrence times of a predicate, we can request the approach to return the occurrence probability once a single occurrence is detected. We set the occurrence probability threshold to 1 and run the approach. If the returned occurrence times is greater than 0, the occurrence probability of the predicate is 1. Otherwise, the approach returns the occurrence times as 0 and an occurrence probability, which is the result we needed. In this paper, we mainly focus on the problem of occurrence times, and will also further discuss how to determine the occurrence probability of a predicate.

## 5 Solution

We first analyze the challenging issues involved in solving this problem, and then propose a unified algorithm framework as the solution suitable for generic predicates. More details are then illustrated by applying the algorithm framework to simple predicates, simple sequences, and interval-constrained sequences.

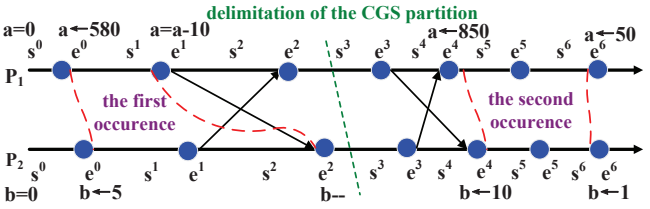


Fig. 4: An example of the locality property: space-time diagram. To detect the predicate $a > 580 \bigwedge 4.4 \leq b \leq 17$

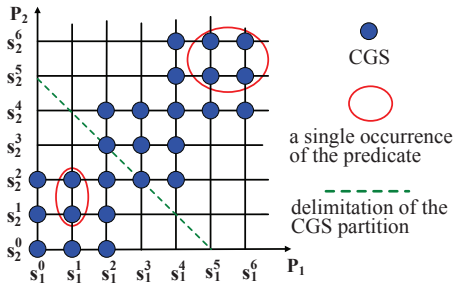


Fig. 5: An example of the locality property: the lattice corresponding to Fig.4

### 5.1 Design Rationale

This problem is a combinatorial optimization problem, in which it is necessity to check the occurrence times of the predicate in each observation. However, we have observed that in the real world, each occurrence of a predicate usually spans over states in the vicinity, rather than states far away from each other. We call this property the *locality property*. This property is described below, and can simplify the solution.

*Property 1* (Locality Property): Each occurrence of a predicate can be detected in states in the vicinity.

Accordingly, in a distributed system, each occurrence of a predicate spans over CGSs in the vicinity and these CGSs are similar in different observations. Therefore, CGSs can be partitioned into different sets, with one set for a single occurrence. If message exchanges take place sufficiently frequently, the partitions will be similar in different observations. For example, in Fig. 4 and Fig. 5, which show the space-time diagram and the lattice for an explosion alarm that was mentioned at the beginning of this paper, it can be seen that each occurrence of the predicate only spans over several neighboring CGSs, and that a common partition into two subsets of CGSs applies to all observations. Utilizing this property, we can simplify the solution by constructing a proper CGS partition for all of the observations and avoiding enumerating them.

A common CGS partition for all of the observations should satisfy the condition that all of the observations must go through each element of it. Otherwise,

some elements may be circumvented. The following definition specifies this condition:

*Definition 1*: A non-concurrent partition of CGS set $W$ is a collection of CGS sub-sets:

$\{P_i(i = 1...n)|P_i(i = 1...n)$ *is a partition of* $W$ *and* $\neg\exists t, k : t \in P_i \bigwedge k \in P_j \bigwedge i \neq j \bigwedge len(t) = len(k)\}$

where $len(t)$ is the number of CGSs from the initial CGS to $t$ following any observation.

The problem is then transferred to determine a non-concurrent partition of CGSs with maximum cardinality, where the real program execution goes through each element of the partition with a probability of at least $p$. We analyze the effectiveness of the idea when $p = 1$ and $p < 1$. When $p = 1$ (i.e., under the definitely modality) and the locality property holds, the detected occurrence times are equal to those of the original problem. When $p < 1$ and the locality property holds, the occurrence times are greater than the real value. For instance, consider a case in which CGSs are partitioned into two subsets and each subset makes the predicate hold once. Assuming that the predicate holds in the observations $\Phi_1$ in the first element, and $\Phi_2$ in the second element, $\sum_{o\in\Phi_1} prob(O) > p$ and $\sum_{o\in\Phi_2} prob(O) > p$, the occurrence times that are detected are 2. In fact, the predicate holds twice only in the common observations $\Phi_1 \cap \Phi_2$, and $\sum_{o\in\Phi_1\cap\Phi_2} prob(O)$ is not necessarily greater than $p$. Let the deviation between the real occurrence times and the value obtained by this approach be $\Delta num$. For a given case, $\Delta num$ with a larger $p$ is less than or equal to $\Delta num$ with a smaller $p$, because more common observations are considered in different partitions. The extreme case is that when $p = 1$, all of the observations in each partition need to be considered.

The last point is how to determine such partitions of CGSs. It can be seen that each element of a partition is determined by its beginning CGSs and ending CGSs, defined as follows:

*Definition 2*: The beginning CGSs of CGS set $W$, denoted by *beginCGS*($W$), are the CGSs that no other CGSs in $W$ can precede:

$\{C_1|C_1 \in W$ *and* $\forall C_2 : C_2 \in W, C_2 \neq C_1 :: \neg(C_2 \rightsquigarrow C_1)\}$

*Definition 3*: The ending CGSs of CGS set $W$, denoted by *endCGS*($W$), are CGSs that cannot precede any CGS in $W$:

$\{C_1|C_1 \in W$ *and* $\forall C_2 : C_2 \in W, C_2 \neq C_1 :: \neg(C_1 \rightsquigarrow C_2)\}$

Clearly, the CGSs in the beginning CGSs or ending CGSs are pairwise concurrent. We also define the paths included in a CGS set $C$:

*Definition 4*: The paths included in CGS set $W$, denoted by $path(W)$ or $path$(*beginCGS*($W$), *endCGS*($W$)), are a set of CGS sequences $C_1, C_2.....C_n$:

$\{C_1, C_2...C_n|C_1 \in$ *beginCGS*$(W), C_n \in$ *endCGS*$(W),$
$C_i \in W(i = 1...n), C_i \prec C_{i+1}(i = 1...n\text{-}1)\}$

Our task is to determine the partition by addressing the following two issues:

*I1*: Determining a set of CGSs that are pairwise concurrent and across all observations to form the beginning or ending CGSs of an element in the partition.

*I2*: Calculating the occurrence probability for each element in the partition.

*I1* guarantees that the result is a non-concurrent partition, and *I2* meets the requirement of occurrence probability. Both issues can be addressed based on lattice. The goal is to determine cuts of a lattice such that the paths between two consecutive cuts, in which the predicate is true, have an occurrence probability of at least $p$. The CGSs in a cut are concurrent with each other and across all observations (*I1*). The occurrence probability between two consecutive cuts, say $T_1$ and $T_2$, can be computed by accumulating the transition probability in $path(T_1, T_2)$, which can be done step by step without redundancy (*I2*).

Besides the correctness of the solution as discussed above, we want to reduce the state space to be explored in order to save on computation (the worst-case time complexity, however, has not been changed). We have the following property:

*Property 2* (Elimination Property): A CGS can be eliminated from further checks if and only if the specified predicate is detected or cannot be detected in all of the paths going through this CGS, and this result cannot be impacted by any following CGS.

Finally, we want a solution that can be adapted to various types of predicates in an efficient manner. We will discuss details of the design in later sections.

### 5.2 Algorithm Framework for Predicate Detection

Since there are various types of predicates and designing a detection algorithm for each of them is tedious, we propose a unified algorithm framework to determine the occurrence times of a predicate subject to an occurrence probability. The framework is shown in Algorithm 1.

We initiate several variables in lines 1-4. *occurTimes* denotes the predicate's occurrence times detected so far. *iniState* denotes the initial CGS of the program execution. For each occurrence of the predicate, *verifiedProb* records the accumulated occurrence probability, and *verifiedSet* records some special CGSs whose following CGSs no longer need to be explored. The main body of the algorithm is lines 5-22. First, the initial CGS *iniState* is configured using Function *iniConfig* (line 5). Then, the CGSs immediately following *iniState* are obtained using Function *extend* and put into *current* (line 6). In lines 7-21, there is a loop calculating the occurrence times subject to an occurrence probability. In each iteration of the loop, the CGSs of *current* are checked against the predicate, and the occurrence probability is accumulated to *verifiedProb* using Function *accumulateProb* (line 9). Function *verifyState* adds some CGSs into *verifiedSet* if needed (line 10). If *verifiedProb* is greater than the given occurrence

**Algorithm 1:** Algorithm framework for determining the occurrence times of a predicate subject to an occurrence probability

```
   Function: detectPredicate(Predicate e, Double occurProb)
1  int occurTimes = 0
2  CGS iniState = C_ini
3  Double verifiedProb = 0
4  Set verfiedSet = ∅
5  iniConfig(iniState)
6  Set current = extend(iniState)
7  while current ≠ {C_final} do
8      foreach C_i ∈ current do
9          verifiedProb=verifiedProb+accumulateProb(e, C_i);
10         verifyState(e, verifiedSet, C_i)
11         if verifiedProb ≥ occurProb then
12             occurTimes = occurTimes + 1
13             restartPredicateDetection(current)
14             verifiedProb = 0
15             verifiedSet = ∅
16             break;
17         end
18     endfch
19     current = elimilateStates(current, verifiedSet)
20     current = extend (current)
21 end
22 return occurTimes
```

probability (line 11), *occurTimes* increases (line 12). We then reset the CGSs in *current* using Function *restartPredicateDetection* and initiate related variables (lines 13-15). After that, the execution jumps out from the loop (line 16). Post-processing is undertaken in lines 19-20. Some CGSs in *current* are eliminated in order to reduce the number of CGSs to be checked (line 19). Finally, *current* is updated for a new iteration of the loop (line 20). The loop terminates when the final CGS $C_{\text{final}}$ is reached (line 7).

This framework is general and applies to different types of predicates. Six functions need to be implemented for each type of predicate: *iniConfig*, *extend*, *restartPredicateDetection*, *elimilateStates*, *accumulateProb*, and *verifyState*. We give a basic implementation of the first four functions in Algorithm 2, while the last two are predicate-specific.

In the basic implementation, we introduce the concept of residual probability, denoted by *residualProb*, for a CGS. It represents the probability that a path from the initial CGS will go through this CGS, excluding the impact of verified CGSs. We do not consider the verified CGSs since the predicate has already been detected or can no longer be detected in the paths that involve them. The purpose of this design is to reduce the state space to be explored in the lattice to speed up the computation.

In Algorithm 2, we initiate the *residualProb* of *inistate* to 1 in Function *iniconfig*. In Function *extend*, the following CGSs of *current* are obtained, and their *residualProb* are then calculated based on the residual probability of *current* and the corresponding transition probability.

In Function *elimilateStates*, the CGSs in *verifiedSet* are removed from *current* for a further check. Property 2 guides the setting of *verifiedSet* in Function *accumulateProb*, and its detailed implementation will be shown in later subsections.

**Algorithm 2:** Basic implementation of the algorithm framework

```
   Function: iniConfig(CGS iniState)
1  iniState.residualProb = 1

   Function: extend(Set current)
1  Set next = ∅
2  foreach C_i ∈ current do
3      foreach C_k such that C_i ≺ C_k do
4          C_k.residualProb =
           C_k.residualProb + C_i.residualProb × tranProb(C_i, C_k)
5          next = next ∪ {C_k}
6      endfch
7  endfch
8  return next

   Function: restartPredicateDetection(Set current)
1  find the minimal CGS set extCurrent such that
   extCurrent||current and extCurrent ∪ current form a cut of the
   lattice
2  current = extCurrent ∪ current
3  processing = verifiedSet
4  while processing ≠ current do
5      Set next = ∅
6      foreach C_i ∈ processing do
7          foreach C_k such that C_i ≺ C_k do
8              C_k.residualProb = C_k.residualProb +
               C_i.residualProb × tranProb(C_i, C_k)
9              next = next ∪ {C_k}
10         endfch
11     endfch
12     processing=next
13 end

   Function: elimilateStates(Set current, Set verifiedSet)
1  current = current − verifiedSet
2  return current
```

In Function *restartPredicateDetection*, a cut is determined to be the ending cut of current detection. Since all of the CGSs in *current* are concurrent, the problem is to find the earliest other CGSs that are concurrent with *current* to form a cut (lines 1-2). Assuming that a CGS ($S_1^x$, $S_2^y$) is included in *current*, we have $extCurrent = \{(S_1^m, S_2^n) | m + n = x + y\} - current$. The sum of the occurrence probabilities of these CGSs should be 1, ensuring that any path starting from the initial CGS reaches this cut before we begin another predicate detection. To achieve this, the residual probability of the cut also takes the residual probabilities coming from *verifiedSet* into account (lines 3-13).

We remove the CGSs in *verifiedSet* for further consideration during the detection, and recover them when initiating a new detection. We have this design mainly for two reasons: First, we make the algorithm also suitable for calculating the occurrence probability of a predicate. In that case, Function *restartPredicateDetection* is not called; hence eliminating the CGSs in *verifiedSet* can save on computations (the amount saved on computations depends on specific applications). Second, the algorithm is efficient if the predicate is false in most situations. This is common in many applications where the users are interested only in the exceptions of a program execution. Our framework supports the optimization for these cases.

In the following three sub-sections, we demonstrate

**Algorithm 3:** Detection of simple predicates

**Function**: accumulateProb(Predicate $e$, CGS $C_p$)
1 **if** $C_p$ satisfies $e$ **then**
2 | *result* = $C_p$.*residualProb*
3 **else**
4 | *result* = 0
5 **end**
6 **return** *result*

**Function**: verifyState(Predicate $e$, Set *verifiedSet*, CGS $C_p$)
1 **if** $C_p$ satisfies $e$ **then**
2 | *verifiedSet* = *verifiedSet* $\cup \{C_p\}$
3 **end**

**Algorithm 4:** Detection of simple sequences

**Function**: accumulateProb(Predicate $e$, CGS $C_p$)
1 **foreach** $C_q$ such that $C_q \prec C_p$ **do**
2 | **for** $i$=1 to $m$ **do**
3 | | $C_p$.*subPredicates*[$i$].*prob* = $C_p$.*subPredicates*[$i$].*prob* + $C_q$.*subPredicates*[$i$].*prob* $\times$ *tranProb*($C_q, C_p$)
4 | **end**
5 **endfch**
6 **if** $C_p$ satisfies $e[1]$ && $\sum_{i=1}^{m} C_p$.*subPredicates*[$i$].*prob* $< C_p$.*residualProb* **then**
7 | $C_p$.*subPredicates*[1].*prob* = $C_p$.*residualProb* $- \sum_{i=2}^{m} C_p$.*subPredicates*[$i$].*prob*
8 **end**
9 **for** $i$=2 to $m$ **do**
10 | **if** $C_p$ satisfies $e[i]$ && $C_p$.*subPredicates*[$i-1$] $> 0$ **then**
11 | | $C_p$.*subPredicates*[$i$].*prob* = $C_p$.*subPredicates*[$i$].*prob* + $C_p$.*subPredicates*[$i-1$].*prob*
12 | | $C_p$.*subPredicates*[$i-1$].*prob* = 0
13 | **end**
14 **end**
15 **return** $C_p$.*subPredicates*[$m$].*prob*

**Function**: verifyState(Predicate $e$, Set *verifiedSet*, CGS $C_p$)
1 **if** $C_p$.*subPredicates*[$m$].*prob* = $C_p$.*residualProb* **then**
2 | *verifiedSet* = *verifiedSet* $\cup \{C_p\}$
3 **end**

the effectiveness of this algorithm framework using three typical types of predicates.

### 5.3 Detection of Simple Predicates

Based on the algorithm framework and the basic implementation shown in Algorithm 1 and 2, the detection of simple predicates is a quite simple process. The detailed algorithm is shown in Algorithm 3.

It includes the implementation of two functions, *accumulateProb* and *verifyState*. Since simple predicates are detected at individual CGSs, the occurrence probability can be accumulated directly. The residual probability is simply returned by Function *accumulateProb*. For Function *verifyState*, if a CGS makes the predicate true, it is marked as a verified CGS, which will be eliminated from further processing (Function *elimilateStates* of Algorithm 2). This is because it satisfies Property 2, considering that if a predicate is detected at one CGS, this fact cannot be changed by the following CGSs.

### 5.4 Detection of Simple Sequences

We use an array $e$ to represent a simple sequence $e_1; e_2; ....e_m$, where $e[i]$ denotes $e_i$. For each CGS, we define an array *subPredicates* to record the detection results, where *subPredicates*[*i*] corresponds to $e[i]$. The $prob$ of *subPredicates*[*i*] records the occurrence probability of $e[i]$ accumulated from the initial CGS to the current CGS.

The detailed algorithm is shown in Algorithm 4. In Function *accumulateProb*, the residual probability of CGS $C_p$ is calculated based on its preceding CGSs. In lines 2-4, the residual probabilities are transferred from the preceding CGSs to $C_p$, taking transition probabilities into account. This is necessary because the observations involving $C_p$'s preceding CGSs only go through $C_p$ with certain transition probabilities. In lines 5-13, the predicate is checked at $C_p$. The first step is to check whether $\sum_{i=1}^{m} C_p$.*subPredicates*[$i$].*prob* $< C_p$.*residualProb*. If the inequality holds, it is known that $e[1]$ is not true in some paths going through $C_p$. Then, $e[1]$ will become true in these paths if $C_p$ satisfies $e[1]$. We add the probability into $C_p$.*subPredicates*[1].*prob*, as in lines 5-7. In lines 8-13, the detection of $e[i]$ is based on the detection of $e[i-1]$. If $C_p$ makes $e[i]$ true, it transfers the residual probability of $e[i-1]$ to $e[i]$ $(2 \leq i \leq m)$. Finally, the function returns $C_p$.*subPredicates*[$m$].*prob*, denoting the accumulated occurrence probability of the whole predicate.

In Function *verifyState*, the CGSs satisfying Property 2 are eliminated from further processing. For a simple sequence, when $e[1]...e[m-1]$ is detected at a CGS, there is no guarantee that the whole predicate will be detected. Only when $e[m]$ is detected at the CGS can the conclusion be reached, which will not be changed by the following CGSs. It is noted that there are multiple observations going through one CGS, so that a simple sequence can be true in some of the observations, but not in others. In such a case, the CGS cannot be eliminated. The exploration of CGS $C_p$ is stopped when $C_p$.*subPredicates*[$m$].*prob* $= C_p$.*residusalProb*. This means that the predicate holds in all of the paths passing through $C_p$.

### 5.5 Detection of Interval-constrained Sequences

An interval-constrained sequence, in the form of $[\theta_1]e_1; ...[\theta_m]e_m; [\theta_m + 1]$, can be viewed as a simple sequence with additional specifications of undesirable states in some intervals. We still use an array $e$ to describe the predicate where $e[i].pos$ and $e[i].neg$ are used to denote $e_i$ and $[\theta_i]$, respectively. $e[m+1].pos$ is not defined. The array *subPredicates* is used to record the detection results. For example, *subPredicates*[1] records the detection result of $[\theta_1]e_1$, and *subPredicates*[2] records the detection result of $[\theta_1]e_1; [\theta_2]e_2$, and so on. Since an interval-constrained sequence includes both positive predicates and negative predicates, we use $C_p$.*failProb* to denote the probability that the paths going through $C_p$ can never make the whole predicate true.

**Algorithm 5:** Detection of interval-constrained sequences

**Function**: accumulateProb(Predicate $e$, CGS $C_p$)
1 **foreach** $C_q$ such that $C_q \prec C_p$ **do**
2  **for** $i$=1 to $m$ **do**
3  $C_p.subPredicates[i].prob = C_p.subPredicates[i].prob + C_q.subPredicates[i].prob \times tranProb(C_q, C_p)$
4  **end**
5 **endfch**
6 **for** $i$=2 to $m$ **do**
7  **if** $C_p$ satisfies $e[i].neg$ **then**
8  $C_p.failProb = C_p.failProb + C_p.subPredicates[i-1].prob$
9  $C_p.subPredicates[i-1].prob = 0$
10  **else if** $C_p$ satisfies $e[i].pos$ **then**
11  $C_p.subPredicates[i].prob = C_p.subPredicates[i].prob + C_p.subPredicates[i-1].prob$
12  $C_p.subPredicates[i-1].prob = 0$
13  **end**
14 **end**
15 **if** $C_p$ satisfies $e[1].neg$ **then**
16  $C_p.failProb = C_p.residualProb - \sum_{i=1}^{m} C_p.subPredicates[i].prob$
17 **else if** $C_p$ satisfies $e[1].pos$ **then**
18  $C_p.subPredicates[1].prob = C_p.residualProb - \sum_{i=2}^{m} C_p.subPredicates[i].prob - C_p.failProb$
19 **end**
20 **if** $e[m+1].neg \neq null$ && $C_p$ satisfies $e[m+1].neg$ **then**
21  $C_p.failProb = C_p.failProb + C_p.subPredicates[m].prob$
22  $C_p.subPredicates[m]$.prob $= 0$
23 **end**
24 **return** $C_p.subPredicates[m].prob$

**Function**: verifyState(Predicate $e$, Set *verifiedSet*, CGS $C_p$)
1 **if** $e[m+1].neg = null$ **then**
2  **if** $C_p.subPredicates[m].prob = C_p.residualProb$ **then**
3  $verifiedSet = verifiedSet \cup \{C_p\}$
4  **end**
5 **end**
6 **if** $C_p.failProb = C_p.residualProb$ **then**
7  $verifiedSet = verifiedSet \cup \{C_p\}$
8 **end**

The detailed algorithm is shown in Algorithm 5. In Function *accumulateProb*, the residual probability of CGS $C_p$ is first transferred from its preceding CGSs, taking transition probabilities into account (lines 2-4), which is similar to simple sequences. The interval-constrained sequence has a special feature that it includes both positive predicates and negative predicates. The detailed processing for that is illustrated in lines 5-22. If $e[i-1].pos$ is true and $C_p$ satisfies $e[i].neg$, the predicate cannot be true according to the definition of interval-constrained sequences; hence, $C_p.subPredicates[i-1].prob$ is transferred to $C_p.failProb$ (lines 6-8). If the preceding CGSs satisfy $e[1].neg$ $e[1].pos$...$e[i-1].neg$ $e[i-1].pos$ and $C_p$ satisfies $e[i].pos$, $C_p.subPredicates[i-1].prob$ needs to transfer to $C_p.subPredicates[i].prob$ (lines 9-12). Similar processing is performed for $e[m+1].neg$ (lines 19-22), while there is no $e[m+1].pos$ to be defined as our definition. Special processing is needed for $e[1]$ (lines 14-18). If $C_p$ satisfies $e[1].neg$, the predicate cannot be true in the paths going through $C_p$, with the exception of the paths that already satisfy $e[i](1 \leq i \leq m)$. Therefore, the probability $C_p.residualProb - \sum_{i=1}^{m} C_p.subPredicates[i].prob$ is transferred to $C_p.failProb$ (lines 14-15). Otherwise, if $C_p$ satisfies $e[1].pos$ for the first time, the residual probability is added to $C_p.subPredicates[1]$.

In Function *verifyState*, we consider both positive and negative predicates. If $e[m+1].neg$ does not exist, the predicate detection ends with $e[m].pos$, which is the same with a simple sequence (lines 1-5). If $e[m+1].neg$ exists, it cannot come to any conclusion as to whether or not the predicate is detected, because $e[m+1].neg$ may be satisfied in one of the following CGSs. In this situation, no CGS can be eliminated. However, if a negative predicate is detected in corresponding intervals, the predicate cannot be satisfied. Similar to the analysis of positive predicates, if *failProb* is equal to *residualProb*, the paths going through the current CGS cannot make the predicate hold, and hence can be eliminated (lines 6-8).

## 6 Discussion

We further prove the correctness of the proposed algorithms and discuss some extended topics.

### 6.1 Correctness of the Algorithms

We prove the correctness of the proposed algorithms.

*Theorem 1:* Algorithm 3 detects the maximum occurrence times of a simple predicate subject to an occurrence probability.

*Proof:* The detection process starts from the initial CGS and advances level by level until the final CGS. During this process, the detection of simple predicates is monotonic, which means that if a simple predicate is detected at a CGS, all of the following CGSs in the paths starting from this CGS also admit the detection. With this property, if a simple predicate is detected at a CGS, the occurrence probability can be accumulated to *verifiedProb* and is invariable in any cut of the lattice following this CGS. Therefore, the CGS can be eliminated from further checks and also from the calculation of residual probability. As the processing continues, if *verifiedProb* reaches the given occurrence probability at a cut of the lattice, the predicate can be guaranteed to be detected because all of the paths contributing to *verifiedProb* pass this cut. Since we find the minimal cut, there is no earlier cut that can meet the requirement of occurrence probability. The process is repeated, and *occruenceTimes* finally returns the maximum occurrence times of the predicate subject to the occurrence probability. □

*Theorem 2:* Algorithm 4 detects the maximum occurrence times of a simple sequence subject to an occurrence probability.

*Proof:* For a CGS $C_p$, $C_p.subPredicates[i].prob$ records the residual probability of $e[1]$....$e[i]$ considering the paths starting from the initial CGS and ending at $C_p$. This can be proofed by the induction of the level number in the lattice. For level 1, this holds since $C_p.subPredicates[1].prob$ is calculated based on the initial CGS, and $C_p.subPredicates[i].prob$ $(i > 1)$

is 0. Assume that the statement is true for level $u$, and that $C_q$ in level $u$ precedes $C_p$ in level $u+1$. This means that $C_q.subPredicates[i-1].prob$ records the residual probability of $e[1]....e[i-1]$ considering the paths starting from the initial CGS to $C_q$. Extending the path one step to $C_p$, the residual probability of $e[1]....e[i]$ can be obtained if $C_p$ satisfies $e[i]$ (lines 5-13). Special processing is needed for $e[1]$ and $e[m+1]$, but the idea is the same (lines 14-22).

The detection process starts from the initial CGS and then advances level by level until the final CGS. $C_p.subPredicates$ records all of the possibilities in the paths starting from the initial CGS and ending at $C_p$. $C_p$ can only be eliminated when the predicate is detected in all possibilities. If *verifiedProb* reaches the given occurrence probability at a cut of the lattice, the predicate can be guaranteed to be detected since all of the paths contributing to *verifiedProb* go to this cut. The cut is minimized to guarantee that the maximum number of occurrence times can be determined. □

*Theorem 3:* Algorithm 5 detects the maximum occurrence times of an interval-constrained sequence subject to an occurrence probability.

Due to limited space, we omit the proof of Algorithm 5, the idea of which is similar to the proof of Algorithm 4, but which takes both positive predicates and negative predicates into consideration.

### 6.2 Relations with Definitely Modality and Possibly Modality

In our paper, the *definitely* and *possibly* modalities [11], [12] are generalized to the occurrence probability. With our algorithms, the detection under the *definitely* and *possibly* modalities also can be done, by setting the occurrence probability threshold to 1 (see the discussion in section 4.3). This method may check all of the CGSs in the worst case. If we only want to know whether a predicate is true under the *possibly* modality, a simpler solution is possible: set the occurrence probability to a sufficiently small value and make Function *restartPredicateDetection* simply exit. The processing can immediately be stopped when an occurrence of the predicate is detected, which can save computation time.

## 7 Performance Evaluation

Simulations are carried out to validate the effectiveness of the proposed algorithms. We show the results of our algorithms in different types of predicates, including simple predicates, simple sequences, and interval-constrained sequences.

### 7.1 Simulation Setup

We simulate an asynchronous distributed system that monitors the number of vehicles at two neighboring intersections, A and B, as shown in Fig. 6. Each

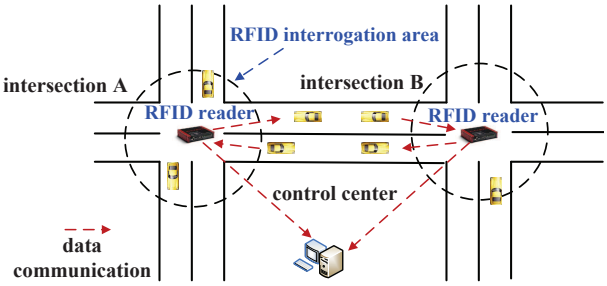


Fig. 6: Simulation scenario

TABLE 1: Parameters adjusted in the simulation

| Parameter | Value |
|---|---|
| *msgInterval* | time intervals at which the readers generate messages |
| *delay* | the mean of the message delays in traveling from one reader to another (following an exponential distribution) |
| *noise* | the mean of noise data collected by the readers (following a normal distribution) |

intersection is equipped with an RFID reader that has no synchronized clock. The two RFID readers communicate with each other via the message piggyback of passing vehicles. Since not all vehicles provide this kind of help and their motion is quite complex, there are different delays in the transmission of messages. The RFID readers report their results to a control center via Wi-Fi, 3G, or vehicle networks.

It is assumed that the vehicles follow a Poisson arrival process to emerge at the intersections. The average number of vehicles arriving at intersection A is 20 and at intersection B is 30. Each vehicle stays at an intersection for several minutes before leaving for other places. The staying time follows an exponential distribution, with a mean of 2 *min* for intersection A and 1 *min* for intersection B. The RFID readers continuously monitor the number of vehicles at the intersections and record the results every minute. An RFID reader sends the messages to the other reader every *msgInterval* minutes to build the happen-before relations. The delay of the messages follows an exponential distribution with a mean of *delay*. The default values of *msgInterval* and *delay* are both 2 *min*. Detection noise is assumed following a normal distribution of $N(noise, \sigma^2)$. We vary these parameters (summarized in TABLE 1) to check the performance of our proposed algorithms. Each data point of the result figures is obtained through 1000 runs of simulations with a confidence interval of 0.95.

### 7.2 Occurrence Times of Simple Predicates

We first check the occurrence times of a simple predicate: *"the total number of vehicles around the two intersections is more than 75"*. The results are shown in Fig. 7-10.

The occurrence times of the predicate is subject to the occurrence probability threshold. We vary it to check the result in Fig. 7. According to the curve

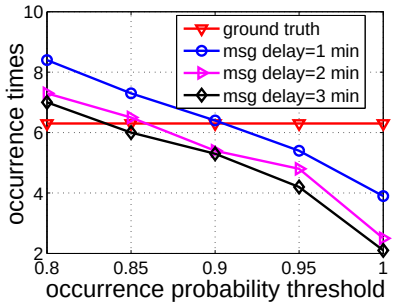


Fig. 7: Detected occurrence times vs. occurrence probability threshold (simple predicate)

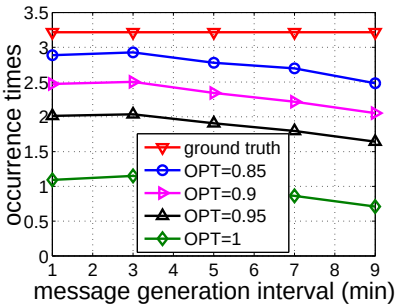


Fig. 8: Detected occurrence times vs. message generation interval (simple predicate)

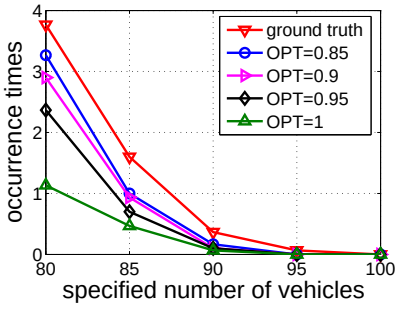


Fig. 9: Detected occurrence times in different ground truths (simple predicate)

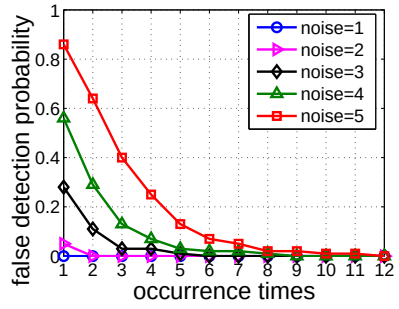


Fig. 10: False detection probability under the definitely modality and considering occurrence times

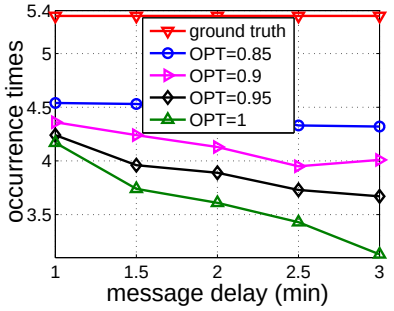


Fig. 11: Detected occurrence times vs. message delay (simple sequence)

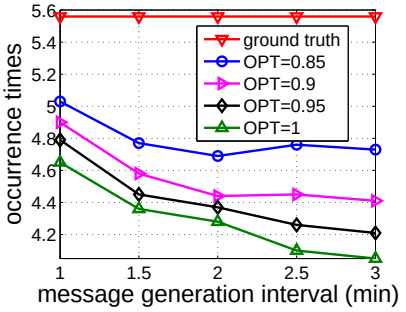


Fig. 12: Detected occurrence times vs. message generation interval (simple sequence)

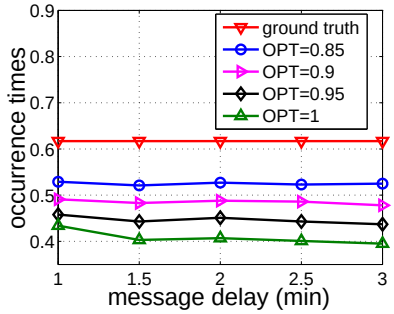


Fig. 13: Detected occurrence times vs. message delay (interval-constrained sequence)

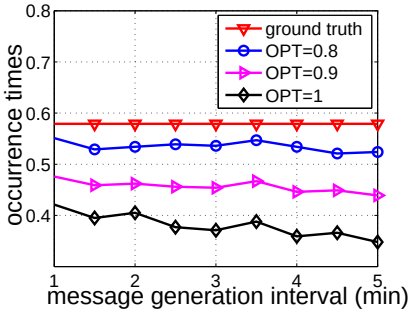


Fig. 14: Detected occurrence times vs. message generation interval (interval-constrained sequence)

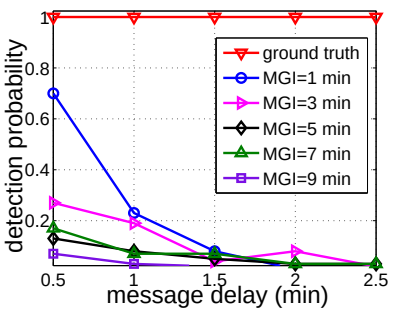


Fig. 15: The detection result of a simple predicate under the definitely modality, with different message delays and generation intervals

OPT: occurrence probability threshold; MGI: message generation interval

labeled with a message delay of 1 *min*, the detected occurrence times of the predicate decreases when the occurrence probability threshold increases. For the detection under the *definitely* modality (i.e., occurrence probability threshold = 1), we can see that the result deviates greatly from the ground truth (by about 38.1%). When we gradually relax the occurrence probability threshold from 1 to 0.9, the result approximates the ground truth. It is noticed that the probability threshold should not be over-relaxed to become less than 0.9, as this causes false-positive results. How to choose a proper occurrence probability threshold is an interesting topic for future research. We further adjust the message delay from 1 *min* to 2 *min* and 3 *min*, and obtain similar results. As the message delay increases, the detected occurrence times decreases, because a larger message delay leads to more CGSs and causes more uncertainty, hence making it difficult to detect the occurrences of the predicate with accuracy.

In Fig. 8, we vary the message generation interval of the RFID readers to check the result. The occurrence times of the predicate decreases when the message generation interval increases. The reason for this is similar to that of the message delay; a larger message generation interval leads to a larger number of possible CGSs, which makes the detection difficult.

In Fig. 9, we check the result by changing the real occurrence times of the predicate (i.e., the ground truth). We adjust the total number of vehicles specified in the predicate from 80 to 100, denoting that the predicate holds more and more rarely. It can be seen that the detected occurrence times with different oc-

currence probability thresholds all decrease. However, the result with an occurrence probability threshold of 0.85 is closer to the ground truth than that with a larger value. It shows that a proper occurrence probability threshold is more reasonable than the *definitely* modality (i.e., occurrence probability threshold = 1).

After we check the change in occurrence times against different factors, we justify the usage of occurrence times taking into consideration the noisy results from RFID readers. The noise is assumed following a normal distributed $N(noise, 10)$. We set the total number of vehicles specified in the predicate to the actual maximum value plus 20 (which means that the specified predicate is never true), to check the probability of false detection in different approaches. This average noise is changed from 1 to 5, and the occurrence probability threshold is set to 0.95. The result is shown in Fig. 10. We can see that if we consider only a single occurrence of the predicate, that the predicate holds or not, the probability of false detection is quite high. Although the specified number of vehicles is never reached in the real world, the probability that the predicate is claimed to be true is much more than 0 (e.g., up to 0.86 when $noise = 5$, and 0.28 when $noise = 3$). To obtain a more convincing result, we need to consider the occurrence times of the predicate. According to the figure, the false detection probability can be reduced when the considered occurrence times increases. The result is even more obvious when the noise level is high. When $noise = 5$, the false detection probability is 0.13 if 5 times of occurrences are considered, and 0.02 if 8 times of occurrences are considered. These results are much better than those where only a single occurrence of the predicate was considered. This justifies the importance and usefulness of predicate detection that takes occurrence times into consideration.

### 7.3 Occurrence Times of Simple Sequences

We then consider simple sequences. Supposing that we want to investigate the relationship between the crowd status of the two intersections, we define a simple sequence $e_1; e_2$, where $e_1$ denotes that "*the number of vehicles around intersection A is more than 30*", and $e_2$ denotes that "*the number of vehicles around intersection B is more than 35 and at the same time the total number of vehicles around these two intersections is more than 60*". We check the effectiveness of our algorithm, and the results are shown in Fig. 11 and Fig. 12.

We first check the occurrence times of the predicate under different message delays. According to Fig. 11, the detected occurrence times decreases when the message delay increases. We also see that the result of our approach is quite close to the ground truth. Considering the case with a message delay of 3 *min*, when the occurrence probability threshold is 0.85, the derivation is bounded to 19.3%, while under the *definitely* modality (i.e., the occurrence probability threshold = 1), the derivation is 41.5%. After that, we change the message generation interval to further check the performance. As shown in Fig. 12, the detected occurrence times decreases when the message generation interval increases. The reason for this is similar to that for message delay. Again, detecting a predicate based on its occurrence times subject to an occurrence probability is more effective than that under the *definitely* modality.

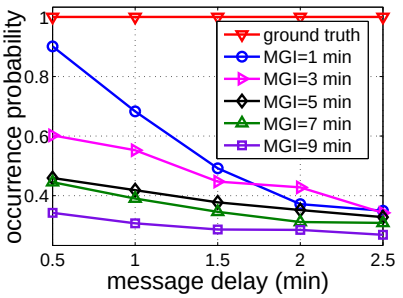


Fig. 16: The occurrence probability of a simple predicate with different message delays and generation intervals

### 7.4 Occurrence Times of Interval-constrained Sequences

Finally, we consider interval-constrained sequences, which specify not only desirable states but also undesirable states. Supposing that we want to investigate a quite crowed situation caused by intersection A, we define an interval-constrained sequence $e_1; [\theta_2]e_2$, where $e_1$ denotes that "th*e number of vehicles around intersection A is more than 40*", $e_2$ denotes that "*the number of vehicles around intersection B is more than 45 and at the same time the total number of vehicles around the two intersections is more than 80*", and $[\theta_2]$ denotes that "*the number of vehicles around the two intersections is not less than 65*". The detection results of our algorithm are shown in Fig. 13 and Fig. 14.

We can see that the results of our algorithm are quite close to the real values when using a proper occurrence probability threshold. According to Fig. 13, assuming that the message delay is 2 *min*, the detected occurrence times is about 85.5% of the real value when the occurrence probability threshold is 0.85 and only 67.0% under the definitely modality (occurrence probability threshold = 1). According to Fig. 14, assuming that the message generation interval is 3 *min*, the detected occurrence times is 93.0% of the real value when the occurrence probability threshold is 0.8, while the comparable figure is 64.0% under the definitely modality. The detected occurrence times decreases when the message delay or message generation interval increases, which is consistent with the results for other types of predicates.

### 7.5 Occurrence Probability of Predicates

Since our algorithm can also be used to determine the occurrence probability of a predicate, we check its effectiveness in this sub-section. Usually, it is preferable to determine the occurrence times of a predicate when the predicate holds several times in the real world, and to determine the occurrence probability of a predicate when the predicate rarely holds.

We consider the same simple predicate discussed in section 7.2. The results of the other types of predicates are similar, and thus are omitted. We compare the results of different approaches in Fig. 15 and Fig. 16.

The detection result under the *definitely* modality is shown in Fig. 15. The returned result is binary, "yes" or "no", where "yes" means that the specified predicate definitely holds and "no" means the other cases. According to the figure, given a message delay of 0.5 *min*, the predicate can be detected with a probability of 0.7 when the message generation interval is 1 *min*. This probability is less than 0.3 when the message generation interval is 3 *min*, and decreases quickly when the interval increases. The probability also decreases quickly when the message delay increases. It is noticed that the probability denotes the probability that the predicate is true, while in other cases the predicate is claimed to be false.

The result of our algorithm is different, as shown in Fig. 16. Instead of a binary result, our algorithm always returns a continuous quantity, the occurrence probability. It describes the possibility that the predicate holds. This value decreases with the increase of the message delay and message generation interval, which matches the ground truth. The occurrence probability distinguishes different cases in distributed systems and provides more detailed information.

## 8 Related Work

Predicates are classified into stable predicates and unstable predicates [19], [20]. The detection of stable predicates is relatively simple and can be solved by distributed snapshot approaches [22]. The detection of unstable predicates is more challenging since multiple observations exist. The *definitely* and *possibly* modalities are introduced in [11] for detecting unstable predicates. The lattice serves as a tool for detecting generic predicates, called *relational predicates*, under the *definitely* and *possibly* modalities [11], [12]. As a special class of predicates, *conjunctive predicates*, specified by a conjunctive expression of local states, are also investigated under these two modalities [15], [27], [28]. In [29], a sequence of predicates that denote both preferred and forbidden properties are formally defined and a solution for their detection is proposed. In [23], three typical types of predicates, including simple predicates, simple sequences, and interval-constrained sequences are further studied. Different from the logical clock, an approximately synchronized physical clock is used to detect predicates in distributed systems [30], [31]. In [10], [32], predicate detection is used for real-time context detection, based on the *definitely* modality.

All of the above works detect only the first occurrence of predicates. In [24], an approach is proposed to detect all occurrences of a conjunctive predicate under the *definitely* modality. The latest work on this topic is [9], where all of the occurrences of both conjunctive predicates and relational predicates are considered. It is based on a software logical clock called *strobe clock*, which resides in each process and gets synchronized once a relevant event happens. The events at the processes are checked to determine whether a predicate holds in the physical world, slightly different from the *definitely* modality. The above two works do not refine the *possibly* modality to provide more detailed information. Moreover, the detection is carried out under the *definitely* modality (or a similar modality) rather than under a generic occurrence probability, which is not sufficient in an error-prone distributed environment.

## 9 Conclusion

In this paper, we investigated the problem of detecting predicates in asynchronous distributed systems. Rather than using the *definitely* and *possibly* modalities, we generalized the predicate detection as based on an occurrence probability, and then further based on the occurrence times subject to an occurrence probability, to provide more detailed information to users. A general algorithm framework was proposed to solve this problem for various types of predicates. We carried out a basic implementation of this algorithm framework and, based on this implementation, developed the algorithms for three typical predicates, namely simple predicates, simple sequences, and interval-constrained sequences. The evaluation results show that our algorithms are effective and outperform existing approaches.

## Acknowledgments

This research is supported in part by Outstanding Academic Talents Startup Funds of Wuhan University No. 216-410100004, Hong Kong RGC under ANR/RGC Joint Research Scheme A-PolyU505/12 and Germany/HK Joint Research Scheme G-PolyU508/13.

## References

[1] S. Ji and Z. Cai, "Distributed data collection and its capacity in asynchronous wireless sensor networks," in *Proc. of INFOCOM*, 2012, pp. 2113–2121.

[2] L. Bui, A. Eryilmaz, R. Srikant, and X. Wu, "Joint asynchronous congestion control and distributed scheduling for multi-hop wireless networks," in *Proc. of INFOCOM*, 2006, pp. 1–12.

[3] F. Babich and M. Comisso, "Analysis of asynchronous multi-packet reception in 802.11 distributed wireless networks," in *Global Telecommunications Conference, 2009. GLOBECOM 2009. IEEE*, 2009, pp. 1–6.
[4] C. Van Berkel, M. Josephs, and S. Nowick, "Applications of asynchronous circuits," *Proceedings of the IEEE*, vol. 87, no. 2, pp. 223–233, 1999.
[5] B. Tang, S. Longfield, S. Bhave, and R. Manohar, "A low power asynchronous gps baseband processor," in *Proc. of IEEE International Symposium on Asynchronous Circuits and Systems*, 2012, pp. 33–40.
[6] J. L. Crowley, "Asynchronous control of orientation and displacement in a robot vehicle," in *Proc. of IEEE International Conference on Robotics and Automation*, 1989, pp. 1277–1282.
[7] P. Flocchini, G. Prencipe, N. Santoro, and P. Widmayer, "Gathering of asynchronous robots with limited visibility," *Theoretical Computer Science*, vol. 337, no. 1-3, pp. 147–168, 2005.
[8] Z. Bouzid, M. Potop-Butucaru, and S. Tixeuil, "Optimal byzantine resilient convergence in asynchronous robots networks," in *Proc. of the International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2009, pp. 165–179.
[9] A. D. Kshemkalyani and J. Cao, "Predicate detection in asynchronous pervasive environments," *IEEE Trans. on Computers*, vol. 62, no. 9, pp. 1823–1836, 2013.
[10] Y. Huang, Y. Yang, J. Cao, X. Ma, X. Tao, and J. Lu, "Runtime detection of the concurrency property in asynchronous pervasive computing environments," *IEEE Trans. on Parallel and Distribued Systems*, vol. 23, no. 4, pp. 744–759, 2012.
[11] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proc. of the ACM/OCR Workshop on Parallel and Distributed Debugging*, 1991, pp. 163–173.
[12] K. Marzullo and G. Neiger, "Detection of global state predicates," in *Proc. of the 5th Workshop on Distributed Algorithms*, 1991, pp. 254–272.
[13] H. Lee, C. Wu, and H. Aghajan, "Vision-based user-centric light control for smart environments," *Pervasive and Mobile Computing*, vol. 7, no. 2, pp. 223–240, 2011.
[14] N. Eagle, A. S. Pentland, and D. Lazer, "Inferring friendship network structure by using mobile phone data," *Proc. of the National Academy of Sciences of the United States of America*, vol. 106, no. 36, pp. 15 274–15 278, 2009.
[15] V. K. Garg and B. Waldecker, "Detection of weak unstable predicates in distributed programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 5, no. 3, pp. 299–307, 1994.
[16] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.
[17] O. Babaoğlu and K. Marzullo, *Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms*. ACM Press/Addison-Wesley Publishing Co., 1993.
[18] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: In search of the holy grail," *Distributed Computing*, vol. 7, no. 3, pp. 149–174, 1994.
[19] A. D. Kshemkalyani and M. Singhal, *Distributed computing: principles, algorithms, and systems*. Cambridge Univ. Press, 2008.
[20] M. Raynal, *Distributed Algorithms for Message-passing Systems*. Springer, 2013.
[21] V. K. Garg, *Elements of distributed computing*. Wiley, 2002.
[22] K. M. Chandy and L. Lamport, "Distributed snapshots: detetermining global states of distributed systems," *ACM Trans. on Computer Systems*, vol. 3, no. 1, pp. 63–75, 1985.
[23] Ö. Babaoglu and M. Raynal, "Specification and verification of dynamic properties in distributed computations," *Journal of Parallel and Distributed Computing*, vol. 28, no. 2, pp. 173–185, 1995.
[24] A. D. Kshemkalyani, "Repeated detection of conjunctive predicates in distributed executions," *Information Processing Letters*, vol. 111, no. 9, pp. 447–452, 2011.
[25] ——, "A fine-grained modality classification for global predicates," *IEEE Trans. on Parallel and Distributed Systems*, vol. 14, no. 8, pp. 807–816, 2003.
[26] H. Luo, Y. Liu, and S. K. Das, "Routing correlated data with fusion cost in wireless sensor networks," *IEEE Tran. on Mobile Computing*, vol. 5, no. 11, pp. 1620–1632, 2006.
[27] V. K. Garg and B. Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Trans. on Parallel and Distributed Systems*, vol. 7, no. 12, pp. 1323–1333, 1996.
[28] M. Hurfin, M. Mizuno, M. Raynal, and M. Singhal, "Efficient distributed detection of conjunctions of local predicates," *IEEE Trans. on Software Engineering*, vol. 24, no. 8, pp. 664–677, 1998.
[29] M. Hurfin, N. Plouzeau, and M. Raynal, "Detecting atomic sequences of predicates in distributed computations," in *Proc. of ACM/ONR workshop on Parallel and distributed debugging*, 1993, pp. 32–42.
[30] J. Mayo and P. Kearns, "Global predicates in rough real time," in *Proc. of the 7th IEEE Symposium on Parallel and Distributeed Processing*, 1995, pp. 17–24.
[31] S. D. Stoller, "Detecting global predicates in distributed systems with clocks," *Distributed Computing*, vol. 13, no. 2, pp. 85–98, 2000.
[32] H. Wei, Y. Huang, J. Cao, X. Ma, and J. Lu, "Formal specification and runtime detection of temporal properties for asynchronous context," in *Proc. of Percom*, 2012, pp. 30–38.

**Weiping Zhu** is currently an associate professor of the International School of Software at Wuhan University. He received the BSc degree from Chongqing University, China, in 2003, the MSc degree from Shanghai Jiao-Tong University, China, in 2006, and the PhD degree from Hong Kong Polytechnic University, Hong Kong, in 2013, all in computer science. Before current position, he worked as a postdoctoral fellow for Hong Kong Polytechnic University, and a technical officer for The University of Hong Kong. He also visited IRISA-INRIA, France, from Nov. 2011 to May. 2012. His research interest includes RFID, WSN, distributed computing, and pervasive computing.

**Jiannong Cao** is currently a chair professor and the head of the Department of Computing at Hong Kong Polytechnic University. He received the BSc degree from Nanjing University, China, in 1982, and the MSc and PhD degrees from Washington State University, USA, in 1986 and 1990, all in computer science. His research interests include parallel and distributed computing, computer networks, mobile and pervasive computing, fault tolerance, and middleware. He co-authored 4 books, coedited 9 books, and published more than 300 technical papers in major international journals and conference proceedings. He has directed and participated in numerous research and development projects and, as a principal investigator, obtained over HK$25 million grants. He is the Chair of Technical Committee on Distributed Computing, IEEE Computer Society, a fellow of IEEE, a member of ACM, and a senior member of China Computer Federation. He has served as an associate editor and a member of editorial boards of many international journals, and a chair and a member of organizing / program committees for many international conferences.

**Michel Raynal** is a professor of computer science at IRISA-INRIA, Rennes, France. His main research interests are the basic principles of distributed computing systems. He is a world leading researcher in the domain of distributed computing. He is the author of numerous papers on distributed computing (more than 120 in journals and 250 papers in intl conferences) and is well-known for his distributed algorithms and his nine books on distributed computing. He has chaired the program committee of the major conferences on the topic (e.g., ICDCS, DISC, SIROCCO, and OPODIS). He has also served on the program committees of many international conferences, and is the recipient of several "Best Paper" awards (ICDCS 1999, 2000 and 2001, SSS 2009, Europar 2010). His h-index is 45. Since 2010, he has been a senior member to the prestigious "Institut Universitaire de France".