Formal Specification and Runtime Detection of Temporal Properties for Asynchronous Context

Hengfeng Wei[1,2], Yu Huang[1,2*], Jiannong Cao[3], Xiaoxing Ma[1,2], Jian Lu[1,2]
[1]*State Key Laboratory for Novel Software Technology*
Nanjing University, Nanjing 210093, China
[2]*Institute of Computer Software, Nanjing University, Nanjing 210093, China*
hengxin0912@gmail.com, {yuhuang, xxm, lj}@nju.edu.cn
[3]*Internet and Mobile Computing Lab, Department of Computing*
Hong Kong Polytechnic University, Hong Kong, China
csjcao@comp.polyu.edu.hk

***Abstract*—Formal specification and runtime detection of temporal properties for pervasive context is one of the primary approaches to achieving context-awareness. Though temporal logics have been widely used in specification of temporal properties, they are faced with severe challenges in Pervasive Computing (PvC) scenarios. First, temporal logics are traditionally defined over infinite traces of possible system behavior. However in PvC scenarios, applications observe finite prefixes of (potentially infinite) traces of environment state evolution, and adapt their behavior accordingly. Second, specification and detection of temporal properties are challenged by the intrinsic asynchrony of PvC environments. Discussions above necessitate a systematic approach to formal specification and runtime detection of temporal properties for asynchronous context. To this end, we propose CTL$_3$ (3-valued Computation Tree Logic), which i) adopts 3-valued semantics to capture the inconclusiveness when applications only observe finite prefixes of environment state evolution; ii) inherits the notion of branching time to capture the uncertainty resulting from the asynchrony of PvC environments. A case study is conducted to demonstrate how CTL$_3$ supports context-awareness in PvC scenarios. The runtime checking algorithm of CTL$_3$ is implemented and evaluated over MIPA - the open-source context-aware middleware we developed. The case study demonstrates the necessity of adopting CTL$_3$ in PvC scenarios, while the performance measurements show the cost-effectiveness of runtime checking contextual properties in CTL$_3$.**

***Keywords*-contextual property; 3-valued semantics; branching time**

I. Introduction

Computing is moving toward pervasive environments where devices and software entities are expected to seamlessly integrate and cooperate to ease users' daily behavior [1], [2]. In pervasive computing (PvC), environmental information is categorized as contexts [3], and applications rely on contexts to adapt their behavior for the users [4], [5].

Runtime detection of contextual properties is one of the primary approaches to achieving context-awareness [3], [6]. For example in an elderly care scenario (which is used as the motivating example throughout the paper), the application may specify property Φ_{elder}: *the elder should not go to bed until he takes the medicine*. Thus, an alarm can be sent when Φ_{elder} is violated.

The contextual properties are often expressed in formal specification languages, e.g., first-order logic [3], [6]. Formal specification of contextual properties enables flexible and unambiguous description of the applications' concerns about the PvC environment [7]. Moreover, formal specification enables automatic checking of specified properties, thus facilitating context-awareness.

Among various contextual properties the application may specify, temporal properties, which delineate temporal evolution of the PvC environment, are of great importance. This is mainly due to dynamism of the pervasive computing environment. In order to cope with this dynamism, the context-aware application needs to keep a stretch of environment state evolution in sight, and use temporal properties to delineate its concerns on the evolution. In the example above, the application is not concerned of specific status of the elder (e.g., "go to bed" or "take medicine"). Rather, it is concerned with the temporal evolution of the elder's status that he should not go to bed until he takes the medicine.

Though temporal logics, such as LTL [8] and CTL [9], are widely used in specification and detection of temporal properties, they are faced with severe challenges in PvC scenarios.

The first challenge is that, classical temporal logics are defined over infinite traces of possible system state evolution. However, in PvC scenarios, the application usually observes finite prefix of the (potentially infinite) trace of environment state evolution, then intelligently adapts its behavior based on observed environment state evolution. Thus we need formal specification schemes which are defined over finite prefixes of environment state evolution.

More importantly, a finite prefix of environment state evolution may not be sufficient to either satisfy or falsify a given contextual property [10]. For example, in the elderly

*Corresponding author.

care scenario above, envision that the elder does not go to bed, and does not take medicine either. In this case, only with more observations can the application decide whether specified property Φ_{elder} is satisfied or violated. According to discussions above, the formal specification of contextual properties needs to provide intuitive and convenient support for the case of being inconclusive.

The second challenge results from the intrinsic asynchrony of PvC environments. Though a few schemes have been proposed to detect contextual properties despite of asynchrony [11], [12], [13], we still need to further discuss how to formally specify temporal properties over the model of asynchronous environment state evolution. Due to the intrinsic uncertainty in asynchronous environments, the application may obtain multiple possible traces of environment state evolution, but does not know which one is the actual trace [14]. Thus, the formal specification of contextual properties should support the notion of branching time [15], to capture the intrinsic uncertainty in asynchronous PvC environments.

Discussions above necessitate a formal specification scheme as well as the runtime detection algorithm for temporal properties of asynchronous context. Toward this objective, we propose CTL$_3$, which i) adopts 3-valued semantics to capture the inconclusiveness when applications only observe finite prefixes of environment state evolution; ii) inherits the notion of branching time in CTL to capture the uncertainty resulting from the asynchrony.

A case study of the elderly care scenario is conducted to demonstrate how CTL$_3$ supports context-awareness in PvC scenarios. The online checking algorithm for CTL$_3$ is implemented and evaluated over MIPA - the open-source context-aware middleware we developed [11], [13], [16]. The case study demonstrates the necessity of adopting 3-valued semantics in PvC scenarios, while the performance measurements show the cost-effectiveness of runtime checking of contextual properties specified in CTL$_3$.

The rest of this paper is organized as follows. In Section II, we discuss the preliminaries. Section III and IV discuss the formal specification and runtime detection of CTL$_3$ properties respectively. Section V and VI present the case study and performance measurements respectively. Section VII discusses the related work. In Section VIII, we conclude the paper with a brief summary and discuss the future work.

II. Preliminaries

In this section, we first discuss how we model temporal evolution of the asynchronous PvC environment, which is the basis for further specification and detection of contextual properties. Then we describe syntax and semantics of classical CTL, based on which we further propose our CTL$_3$.

A. Modeling Asynchronous Environment State Evolution

The detection of contextual properties assumes the availability of an underlying context-aware middleware [3], [11]. The middleware accepts properties from the application, persistently collects context data from multiple sources, and detects at runtime whether the specified properties hold. Specifically, a collection of *non-checker processes* $P^{(1)}, P^{(2)}, \cdots, P^{(n)}$ are deployed to monitor specific regions or aspects of the environment. Typical examples of non-checker processes are software processes manipulating networked sensors. One *checker process* P_{che} is in charge of the runtime detection of contextual properties. P_{che} is usually a third-party service deployed on the middleware.

1) The Lattice of Environment State Evolution: Execution of process $P^{(k)}$ produces its trace, which consists of *local states* connected by *contextual events*: "$s_0^{(k)}, e_0^{(k)}, s_1^{(k)}, e_1^{(k)}, s_2^{(k)}, e_2^{(k)}, \ldots$". The contextual events may be local (e.g., update of the context data), or global (e.g., sending/receiving messages).

To cope with the asynchrony, we re-interpret the notion of time based on the classical Lamport's definition of the *happen-before* (denoted by '$\rightarrow$') relation resulting from message passing [17], and its "on-the-fly" coding given by logical vector clocks [18]. Specifically, for two *contextual events*, $e_1 \rightarrow e_2$ iff i) e_1 and e_2 are on the same non-checker process, and e_1 comes before e_2; or ii) e_1 is the sending of a message by one non-checker process and e_2 is the recipient of the same message by another non-checker process; or iii) there exists some e_3 such that $e_1 \rightarrow e_3 \rightarrow e_2$. For two *states*, $s_1 \rightarrow s_2$ iff the ending of s_1 *happen-before* the beginning of s_2 (note that the beginning and ending of a state are both contextual events).

A *global state* $\mathcal{G} = (s^{(1)}, s^{(2)}, \ldots, s^{(n)})$ is defined as a vector of local states from each $P^{(k)}$. $\mathcal{C}$ is a *Consistent Global State (CGS)* if the constituent local states are pairwise concurrent, i.e.,

$$\mathcal{C} = (s^{(1)}, s^{(2)}, \ldots, s^{(n)}), \forall i, j : i \neq j :: \neg(s^{(i)} \rightarrow s^{(j)})$$

Intuitively, the CGS denotes a snapshot or meaningful observation of the asynchronous environment [19].

The *precede* relation (denoted by '$\prec$') between two CGSs $\mathcal{C} \prec \mathcal{C}'$ is defined as: $\mathcal{C}'$ can be obtained via advancing $\mathcal{C}$ by *exactly one* local state on some non-checker process. The *lead-to* relation (denoted by '$\rightsquigarrow$') is the transitive closure of '$\prec$'. The key notion in the modeling is that all CGSs with '$\rightsquigarrow$' form a *lattice* [14], as depicted in Fig. 1.

In order to model temporal evolution of the environment, we need to keep in sight a stretch of meaningful observations of the environment. Thus, we define the *CGS sequence* $\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j)$ as:

$$\mathcal{S}(\mathcal{C}_i, \mathcal{C}_j) = \mathcal{C}_i, \mathcal{C}_{i+1}, \ldots, \mathcal{C}_j (0 \leq i \leq j),$$
$$\forall k : i \leq k \leq j-1 :: \mathcal{C}_k \prec \mathcal{C}_{k+1}$$

We specify predicates over CGSs to delineate static properties concerning specific snapshot of the environment. The predicates over CGSs can be viewed as the labeling of CGSs

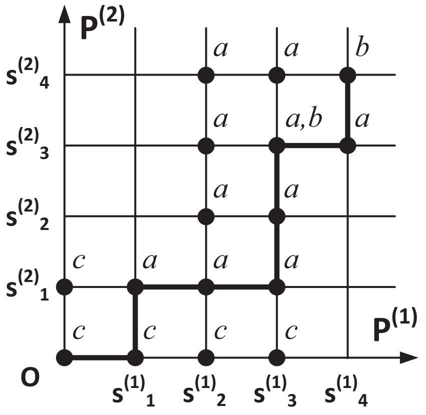

Figure 1. The lattice of CGSs

with letters from a finite alphabet (i.e., all pre-defined CGS predicates). Fig. 1 is an example where each CGS is labeled with the predicates ('a','b' or 'c') it satisfies.

Please refer to our previous work for more discussions on modeling of asynchronous environment state evolution [11], [12], [13], as well as the implementation [16].

2) Treating the Lattice as a Transition System: Essentially, the lattice of CGSs can be regarded as an abstract model of environment state evolution [20]. It is known that *transition system* [21] often serves as a typical model for computations. Thus, we treat the lattice as a transition system, which is defined in Definition II.1.

Definition II.1 *Transition system*

A transition system $M = (S, T, I, AP, L)$ [1] is defined as:

- *S is a set of states.*
- *$T \subseteq S \times S$ is a transition relation.*
- *$I \subseteq S$ is a set of initial states.*
- *AP is a set of atomic propositions.*
- *$L : S \to 2^{AP}$ is a labeling of each state with the set of atomic propositions which hold at the state.*

Note that all the five components of the transition system have obvious correspondences with the notations used in the lattice. We further define path of the transition system as:

Definition II.2 *Path of Transition System*

A path π of transition system M is an infinite *sequence of states $\pi = (s_0, s_1, \ldots)$ such that*

$$\forall i \geq 0, (s_i, s_{i+1}) \in T.$$

The set of infinite paths which start with $s_0 = s$ is denoted by $Paths(s)$.

[1]Note that "action" is not of concern here and left undefined.

We use $\pi[i] = s_i$ to denote the $(i + 1)^{th}$ state of $\pi = (s_0, s_1, \ldots)$. In particular, the *finite* sequence of states $(s_0, s_1, \ldots, s_n)$ is called *finite path* and denoted by $\hat{\pi}$. The length of $\hat{\pi} = (s_0, s_1, \ldots, s_n)$ is $|\hat{\pi}| = n$. The set of finite paths of which each element starts with $s_0 = s$ is denoted by $Paths_{fin}(s)$.

Note that as model of computation of reactive system, the transition system is often assumed to have no terminal states [21]. So all the *paths* induced from it are infinite. The major difference between CGS sequence in the lattice and path of transition system is that the former is finite while the latter is infinite. We argue that the difference is of importance and serves as one of the primary motivations of this work (as detailed in Section III).

B. Computation Tree Logic

The syntax of CTL is defined in Definition II.3:

Definition II.3 *Syntax of CTL*

CTL state formulae *over the set AP of atomic proposition are formed according to the following grammar:*

$$\Phi ::= \top \mid a \mid \Phi_1 \wedge \Phi_2 \mid \neg\Phi \mid \exists\varphi \mid \forall\varphi$$

where $a \in AP$ and φ is a path formula.

CTL path formulae *are formed according to the following grammar:*

$$\varphi ::= \bigcirc\Phi \mid \Phi_1 \cup \Phi_2$$

where Φ, Φ_1, and Φ_2 are state formulae.

CTL formulae are interpreted over the states and infinite paths of a transition system (Definition II.1). Specifically,

Definition II.4 *Satisfaction relation for CTL*

Let $a \in AP$ be an atomic proposition, Φ and Ψ be CTL state formulae, and φ be a CTL path formula. The satisfaction relation $\models$ for state formulae *and state s is defined as follows:*

$$\begin{aligned} s \models a &\iff a \in L(s) \\ s \models \neg\Phi &\iff \text{not } (s \models \Phi) \\ s \models \Phi \wedge \Psi &\iff s \models \Phi \wedge s \models \Psi \\ s \models \exists\varphi &\iff \exists\pi \in Paths(s).\pi \models \varphi \\ s \models \forall\varphi &\iff \forall\pi \in Paths(s).\pi \models \varphi \end{aligned}$$

The satisfaction relation $\models$ for path formulae and path π is defined as follows:

$$\begin{aligned} \pi \models \bigcirc\Phi &\iff \pi[1] \models \Phi \\ \pi \models \Phi \cup \Psi &\iff \exists j \geq 0.\big(\pi[j] \models \Psi \\ &\qquad \wedge \forall 0 \leq k < j.(\pi[k] \models \Phi)\big) \end{aligned}$$

In the following Section III, we propose the 3-valued semantics of CTL$_3$ based on the boolean semantics of CTL.

III. Formal Specification of Temporal Properties for Asynchronous Context

We propose CTL$_3$ for the specification of temporal properties for asynchronous context. CTL$_3$ inherits the notion of branching time from CTL. CTL$_3$ has the same syntax with CTL (see Definition II.3), but adopts 3-valued semantics to cope with the case of being inconclusive. We first discuss the motivation of adopting 3-valued semantics, and then present the formal definition.

A. Why 3-valued Semantics: a Motivating Example

The primary motivation of adopting 3-valued semantics arises from the observation that, in PvC scenarios, the application usually first monitors how the environment evolves, and then takes context-aware adaptation accordingly. When the application only observes finite prefix of the potentially infinite environment state evolution, its (limited) observation may not be sufficient to either satisfy or falsify the specified contextual property [10].

Also take the elderly care scenario as an example. Suppose that the set AP of atomic propositions is $\{\text{sleep, medicine}\}$. The temporal property of concern is that "the elder should not go to sleep until he takes the medicine", which can be specified in CTL as:

$$\Phi_{elder} = \forall\big(\neg\text{sleep} \cup \text{medicine}\big)$$

Intuitively, we evaluate Φ_{elder} to be "true" if the elder takes medicine before he goes to bed. Meanwhile, we evaluate Φ_{elder} to be "false" if the elder has gone to bed, but has never taken the medicine.

Consider the situation in which the elder keeps awake without having taken medicine yet. Formally, every CGS in the lattice of CGSs satisfies "$\neg$sleep $\wedge$ $\neg$medicine".

It is obvious to find in the example above that, current observation can neither satisfy nor falsify the property Φ_{elder}. That is to say, the traditional boolean semantics of CTL does not provide intuitive and convenient support for this case of "being inconclusive". However, this case of being inconclusive may often appear in PvC scenarios, since the application needs to decide whether to adapt based on finite and incomplete observations.

Discussions above motivate us to adopt 3-valued semantics for context-aware applications. In CTL$_3$, we just provide a third value "inconclusive" (denoted by '?') for the case of being inconclusive. Based on the 3-valued semantics, we can evaluate Φ_{elder} to be "inconclusive" for the example above.

B. Formal Definition of the 3-valued Semantics in CTL$_3$

The 3-valued semantics of CTL$_3$ is related to the finite paths. We use the symbols '$\top$','$\bot$', and '?' to denote "true", "false", and *"inconclusive"* respectively.

We interpret the connectives '$\neg$' and '$\wedge$' using Kleene's 3-valued propositional logic [22]. Specifically, the 3-valued semantics of the two connectives are defined as follows:

Definition III.1 *3-valued semantics of '$\neg$' and '$\wedge$'*

$$\neg(\top) = \bot, \qquad \neg(\bot) = \top, \qquad \neg(?) = ?$$
$$\top \wedge \bot = \bot, \qquad \top \wedge ? = ?, \qquad \bot \wedge ? = \bot$$

Truth values of a state formula Φ with respect to a state s and a path formula φ with respect to a finite path $\hat{\pi}$ are denoted by $[s \models \Phi]$ and $[\hat{\pi} \models \varphi]$ respectively. The semantics of CTL$_3$ is given in Definition III.2. Intuitively, compared to CTL, the 3-valued semantics of CTL$_3$ explicitly states which is true and which is false.

Definition III.2 *Satisfaction relation for CTL$_3$*

The 3-valued satisfaction relation $\models$ for state formula *and* state s *is as follows:*

$$[s \models a] = \begin{cases} \top & a \in L(s) \\ \bot & o.w. \end{cases}$$
$$[s \models \neg\Phi] = \neg([s \models \Phi]).$$
$$[s \models \Phi \wedge \Psi] = ([s \models \Phi]) \wedge ([s \models \Psi]).$$
$$[s \models \exists\varphi] = \bigvee_{\hat{\pi}} \big([\hat{\pi} \models \varphi] \mid \hat{\pi} \in \ \mathit{Paths}_{fin}(s)\big)$$
$$[s \models \forall\varphi] = \bigwedge_{\hat{\pi}} \big([\hat{\pi} \models \varphi] \mid \hat{\pi} \in \ \mathit{Paths}_{fin}(s)\big)$$

The 3-valued satisfaction relation $\models$ for path formula *and* finite path $\hat{\pi}$ *is defined by*

$$[\hat{\pi} \models \bigcirc\Phi] = \begin{cases} [\hat{\pi}[1] \models \Phi] & |\hat{\pi}| \geq 1 \\ ? & o.w. \end{cases}$$

$$[\hat{\pi} \models \Phi \cup \Psi] = \begin{cases} \top & \exists 0 \leq j \leq |\hat{\pi}|.\big([\hat{\pi}[j] \models \Psi] = \top \\ & \wedge(\forall 0 \leq k \leq j.[\hat{\pi}[k] \models \Phi] = \top)\big) \\ \bot & \forall 0 \leq j \leq |\hat{\pi}|.\big([\hat{\pi}[j] \models \Psi] = \top \\ & \rightarrow \exists 0 \leq k \leq j.[\hat{\pi}[k] \models \Phi] \models \bot\big) \\ ? & o.w.(i.e., \forall 0 \leq j \leq |\hat{\pi}|. \\ & [\hat{\pi}[j] \models \Phi \wedge \neg\Psi] = \top) \end{cases}$$

$$[\hat{\pi} \models \Diamond\Phi] = \begin{cases} \top & \exists 0 \leq j \leq |\hat{\pi}|.([\hat{\pi}[j] \models \Phi] = \top) \\ ? & o.w. \end{cases}$$

$$[\hat{\pi} \models \Box\Phi] = \begin{cases} \bot & \exists 0 \leq j \leq |\hat{\pi}|.([\hat{\pi}[j] \models \Phi] = \bot) \\ ? & o.w. \end{cases}$$

In the following Section IV, we discuss how to check CTL$_3$ formulae at runtime.

IV. Runtime Detection of Temporal Properties for Asynchronous Context

In this section, we propose an online CTL$_3$ checking algorithm. We discuss the design rationale, then present the detailed design and an illustrating example.

A. Checking CTL$_3$ Formulae: Design Rationale

Here we take modality '$\exists\cup$' as an example to show the design rationale. Checking of other modalities (e.g., '$\exists\bigcirc$' and '$\forall\cup$') is principally the same and is omitted due to the limit of space.

The standard CTL checking algorithm relies on the expansion laws for CTL [21], as defined in Definition IV.1.

Definition IV.1 *Expansion Law for '$\exists\cup$' in CTL*
The expansion law for '$\exists\cup$' is

$$\exists(\Phi \cup \Psi) \equiv \Psi \vee \big(\Phi \wedge \exists \bigcirc \exists(\Phi \cup \Psi)\big)$$

Intuitively, the expansion law for $\exists(\Phi \cup \Psi)$ reflects its recursive nature behind. Specifically, $\exists(\Phi\cup\Psi)$ holds in state s if Ψ holds in the *current state* or Φ holds in the *current state* and $\exists(\Phi \cup \Psi)$ holds for *some direct successor state*.

The CTL$_3$ checking algorithm checks, for any state s and state formula Φ, whether $s \models \Phi$ is validated or violated. In standard CTL checking algorithm, $Sat(\Phi)$ is used to denote the set of all states satisfying Φ and is computed recursively based on the above-mentioned expansion laws (Definition IV.1). However, in the CTL$_3$ case, $S \setminus Sat(\Phi)$ does not represent all states *falsifying* Φ due to the existence of the third value '?'. Therefore we have to distinguish "true"('$\top$'), "false"('$\bot$') from "inconclusive"('?') and explicitly identify each truth value of Φ with

$$Sat_\top(\Phi) = \{s \mid [s \models \Phi] = \top\},$$

$$Sat_\bot(\Phi) = \{s \mid [s \models \Phi] = \bot\},$$

and

$$Sat_?(\Phi) = \{s \mid [s \models \Phi] = ?\}.$$

We use notations Post(s) and Post$^*(s)$ to represent the set of direct successor states and the set of states that are reachable from state s respectively. The notations Pre(s) and Pre$^*(s)$ have analogous meanings.

Definition IV.2 *Characterization of $Sat(\cdot)$ for CTL$_3$*
$Sat_\top\big(\exists(\Phi \cup \Psi)\big)$ *is the* smallest *subset $T \subseteq S$ satisfying*

$$Sat_\top(\Psi) \cup \{s \in Sat_\top(\Phi) \mid Post(s) \cap T \neq \emptyset\}$$

on the other hand,
$Sat_\bot\big(\exists(\Phi \cup \Psi)\big)$ *is the* largest *set $T \subseteq S$ satisfying*

$$Sat_\top(\neg\Phi \wedge \neg\Psi) \cup \{s \in Sat_\top(\Phi \wedge \neg\Psi) \mid Post(s) \subseteq T\}$$

and, therefore,

$$Sat_?\big(\exists(\Phi\cup\Psi)\big) = S \setminus Sat_\top\big(\exists(\Phi\cup\Psi)\big) \setminus Sat_\bot\big(\exists(\Phi\cup\Psi)\big).$$

Note that the characterization of $Sat_\bot\big(\exists(\Phi \cup \Psi)\big)$ is mainly based on the following logical inference:

$$\begin{aligned}\neg\exists(\Phi \cup \Psi) &\equiv \forall\big((\Phi \wedge \neg\Psi)\mathbf{W}(\neg\Phi \wedge \neg\Psi)\big)\\ &\equiv (\neg\Phi \wedge \neg\Psi)\\ &\vee \big((\Phi \wedge \neg\Psi) \wedge \forall \bigcirc \forall(\Phi \wedge \neg\Psi)\mathbf{W}(\neg\Phi \wedge \neg\Psi)\big)\end{aligned}$$

where $\mathbf{W}$ (called *weak until*) is defined by:

$$\varphi\mathbf{W}\psi \equiv (\varphi \cup \psi) \vee \Box\varphi$$

B. Checking CTL$_3$ Formulae: Detailed Design

This section is concerned with CTL$_3$ checking algorithm over lattice $\mathbf{L}$. A decision algorithm is proposed to check whether $[\mathbf{L} \models \Phi] = \top$ or $[\mathbf{L} \models \Phi] = \bot$ for given lattice $\mathbf{L}$ and CTL$_3$ state formula Φ according to the following definition:

$$[\mathbf{L} \models \Phi] = \top \iff \mathcal{C}_0 \in Sat_\top(\Phi)$$

$$[\mathbf{L} \models \Phi] = \bot \iff \mathcal{C}_0 \in Sat_\bot(\Phi)$$

where $\mathcal{C}_0$ denotes the unique initial CGS in lattice $\mathbf{L}$.

The basic idea behind is the same with that of standard CTL formulae checking, where the satisfaction sets for all sub-formulae of Φ are computed recursively [21]. The overall algorithm is sketched in Algorithm 1 where $Sub(\Phi)$ is the set of all sub-formulae of Φ.

Algorithm 1 Online checking algorithm for CTL$_3$

```
1: Upon each new CGS is constructed
2: for all $i \leq |\Phi|$ do
3:     for all $\Psi \in Sub(\Phi)$ with $|\Psi| = i$ do
4:         compute $Sat_\top(\Phi), Sat_\bot(\Phi)$, and $Sat_?(\Phi)$
5:     end for
6: end for
```

The online checking algorithm for CTL$_3$ modalities evaluates on the current state and *update (not re-check)* the truth values of state formulae on older states. The update stage consists of two backward propagation processes which are intended to separate "true"('$\top$') and "false"('$\bot$') from "inconclusive"('?').

To better understand the checking algorithm, we take the checking procedure of the modality $\exists(\Phi \cup \Psi)$ as an example. In the algorithm, we maintain $Sat_\top\big(\exists(\Phi \cup \Psi)\big)$, $Sat_\bot\big(\exists(\Phi\cup\Psi)\big)$, and $Sat_?\big(\exists(\Phi\cup\Psi)\big)$ which are mentioned above (Subsection IV-A). In addition, in order to achieve online checking, we use $Sat_{\top_\circ}(\Phi \cup \Psi)$ and $Sat_{\bot_\circ}(\Phi \cup \Psi)$ to denote the new states whose truth values changed from "inconclusive"('?') to "true"('$\top$') or "false"('$\bot$') respectively during the last update.

Pseudo codes for checking the modality '$\exists\cup$' with two arguments Φ and Ψ are listed in Algorithm 2 and Algorithm 3. In the following, the arguments Φ and Ψ will be omitted if they are clear from the context.

Algorithm 2 Checking algorithm for $\exists(\Phi \cup \Psi)$: (part 1)

```
 1: $E := Sat_{\top_\circ}(\Psi)$
 2: $Sat_{\top_\circ}(\exists\cup) := E$
 3: $Sat_\top(\exists\cup) := Sat_\top(\exists\cup) \cup E$
 4: while $E \neq \emptyset$ do
 5:   let $s \in E$
 6:   $E := E \setminus \{s\}$
 7:   for all $s' \in Pre(s) \wedge s' \in Sat_?(\exists\cup)$ do
 8:     if $s' \in Sat_\top(\Phi \wedge \neg\Psi)$ then
 9:       $E := E \cup \{s'\}$
10:       $Sat_\top(\exists\cup) := Sat_\top(\exists\cup) \cup \{s'\}$
11:       $Sat_?(\exists\cup) := Sat_?(\exists\cup) \setminus \{s'\}$
12:       $Sat_{\top_\circ}(\exists\cup) := Sat_{\top_\circ}(\exists\cup) \cup \{s'\}$
13:     end if
14:   end for
15: end while
```

Algorithm 2 which corresponds to the backward update process for $Sat_\top(\exists\cup)$ is explained in detail:

1) E is a set consisting of state s satisfying $[s \models \exists(\Phi \cup \Psi)] = \top$ during the current iteration of checking. E is initialized as $Sat_{\top_\circ}(\Psi)$. State $s \in Sat_{\top_\circ}(\Psi) = E$ satisfies '$\exists\cup$' trivially (as shown in Line 1 - Line 3).
2) The main iterative procedure of the backward update to compute $Sat_\top(\exists\cup)$ is implemented in the *While* loop (as shown in Line 4 - Line 15). Note that the procedure closely corresponds to the characterization of $Sat_\top(\exists\cup)$ in Definition IV.2.

Algorithm 3 which corresponds to the backward update process for $Sat_\perp(\exists\cup)$ is explained in detail:

1) State $s \in Sat_{\top_\circ}(\neg\Phi \wedge \neg\Psi)$ falsifies '$\exists\cup$' trivially. Thus, $Sat_{\top_\circ}(\neg\Phi \wedge \neg\Psi)$ serves as initialization for $Sat_{\perp_\circ}(\exists\cup)$ and $Sat_\perp(\exists\cup)$. E contains any state $s \in Sat_\top(\neg\Phi\wedge\neg\Psi)$ which is not visited. It is initialized as $Sat_{\top_\circ}(\neg\Phi \wedge \neg\Psi)$. T contains any state s which may falsify '$\exists\cup$' and is needed to be checked further (as shown in Line 1 - Line 4).
2) The main iterative procedure of the backward update to compute $Sat_\perp(\exists\cup)$ is implemented in the *While* loop (as shown in Line 5 - Line 20). It corresponds to the characterization of $Sat_\perp(\exists\cup)$ in Definition IV.2. count$[s]$ keeps track of the number of direct successors of s not in E. It is used to test whether Post$(s) \subseteq E$ (as shown in Line 9 - Line 11) [2]. During the backward process, states are iteratively removed from T, for which it has been established that they falsify '$\exists\cup$'. Meanwhile, such states are inserted to E to enable the possible removal of other states in T.

Taking the space for a single CGS as one unit, the worst-case space cost of lattice maintenance is $\mathcal{O}(p^n)$, where p is

[2]The implementation detail related to online characteristic of algorithm is omitted here.

Algorithm 3 Checking algorithm for $\exists(\Phi \cup \Psi)$: (part 2)

```
 1: $Sat\perp_\circ(\exists\cup) := Sat_{\top_\circ}(\neg\Phi \wedge \neg\Psi))$
 2: $Sat_\perp(\exists\cup) := Sat_\perp(\exists\cup) \cup Sat_{\top\circ}(\neg\Phi \wedge \neg\Psi)$
 3: $E := Sat_{\top_\circ}(\neg\Phi \wedge \neg\Psi)$
 4: $T := Sat_\top(\Phi \wedge \neg\Psi)$
 5: while $E \neq \emptyset$ do
 6:   let $s \in E$
 7:   $E := E \setminus \{s\}$
 8:   for all $s' \in \mathrm{Pre}(s) \wedge s' \in Sat_?(\exists\cup)$ do
 9:     if $s' \in T$ then
10:       count$[s']$ := count$[s'] - 1$
11:       if count$[s'] = 0$ then
12:         $T := T \setminus \{s'\}$
13:         $E := E \cup \{s'\}$
14:         $Sat_?(\exists\cup) := Sat_?(\exists\cup) \setminus \{s'\}$
15:         $Sat_{\perp_\circ}(\exists\cup) := Sat_{\perp_\circ}(\exists\cup) \cup \{s'\}$
16:         $Sat_\perp(\exists\cup) := Sat_\perp(\exists\cup) \cup \{s'\}$
17:       end if
18:     end if
19:   end for
20: end while
```

the upper bound of number of local states of non-checker processes, and n is the number of non-checker processes which is usually not too large. The time complexity of standard checking algorithm for CTL is $\mathcal{O}\big((|S|+|T|)\cdot|\Pi|\big)$ [21], where $|S|$ is the number of states of transition system (in this paper, it is the space cost for lattice discussed above), $|T|$ is the number of transitions of transition system (correspondingly, it is the number of edges of lattice), and $|\Pi|$ is the length of CTL formula Π. The *worst-case time complexity* of our online checking algorithm is also $\mathcal{O}\big((|S|+|T|)\cdot|\Pi|\big)$. However, the cost of CTL$_3$ checking usually does not reach the worst-case upper bound. Please refer to experimental evaluations in Section VI.

C. Checking CTL$_3$ Formulae: an Illustrating Example

In order to illustrate the algorithm, we take modality '$\forall\cup$' (which is used in the case study of Section V) as an example. We consider the update stages for $Sat_\top(\forall\cup)$ and $Sat_\perp(\forall\cup)$.

Consider the lattice structure given in Fig. 2 on which we will check $\Phi_{elder} = \forall(\Phi \cup \Psi)$ based on 3-valued semantics with "$\Phi = \neg$sleep" and "$\Psi =$ medicine".

In the process of update for $Sat_\top(\forall\cup)$, $Sat_\top(\Psi) = Sat_\top(\text{medicine}) = \{1,4,8,12,6,10,13,15\}$, and $Sat_\top(\Phi\wedge\neg\Psi) = Sat_\top(\neg\text{sleep} \wedge \neg\text{medicine}) = \{0,2,5,3,7,11\}$. Intuitively, we can make use of $Sat_\top(\Psi)$ to force the states in $Sat_\top(\Phi \wedge \neg\Psi)$ to satisfy '$\forall\cup$'. Meanwhile, it is important to note that the state s satisfies $\forall(\Phi \cup \Psi)$, only if *all* the finite paths from s satisfy $\Phi \cup \Psi$. Therefore, in this example, $Sat_\top(\Psi)$ is not enough to draw conclusion that states in $Sat_\top(\Phi \wedge \neg\Psi)$ satisfy '$\forall\cup$'. On the other hand, for $Sat_\perp(\forall\cup)$, $Sat_\top(\neg\Phi \wedge \neg\Psi) = Sat_\top(\text{sleep} \wedge$

$\neg$medicine$) = \{9, 14\}$, and $Sat_\top(\Phi \wedge \neg\Psi) = Sat_\top(\neg$sleep$\wedge$ $\neg$medicine$) = \{0, 2, 5, 3, 7, 11\}$. Intuitively, we can make use of $Sat_\top(\neg\Phi \wedge \neg\Psi)$ to prevent the states in $Sat_\top(\Phi \wedge \neg\Psi)$ from satisfying '$\forall$U'.

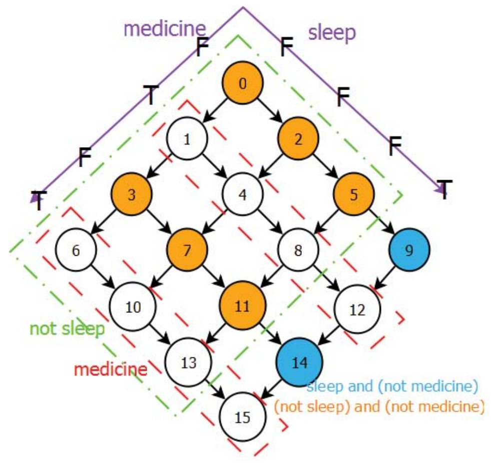

Figure 2. Illustration of checking $\Phi_{elder} = \forall(\neg\text{sleep} \cup \text{medicine})$

V. Case Study

In this section, we conduct a case study to demonstrate how the CTL$_3$ specification and checking algorithm facilitate context-awareness in asynchronous PvC environment. We keep investigating the elderly care scenario, in which an elder should take the medicine before going bed. In this scenario, an application is persistently aware of the elder's status and reminds him of taking medicine in time.

In our scenario, two RFID readers $Reader^{(1)}_{Bed}$ and $Reader^{(2)}_{Med}$ are deployed to detect whether the elder is in the bed or near the medicine box respectively. The elder carries a smart phone, which serves as the sink node of the body-area-network and detects the elder's activities (e.g.,"lying down" or "pouring water into the glass"). Note that the smart phone and the RFID readers do not necessarily have synchronized clocks or shared memory.

A. Modeling of the Temporal Evolution of Environment State

Non-checker processes $P^{(1)}$ and $P^{(2)}$ are deployed on $Reader^{(1)}_{Bed}$ and $Reader^{(2)}_{Med}$ respectively to track locations of the elder persistently. Meanwhile, $P^{(3)}$ and $P^{(4)}$ are deployed on the mobile phone to detect activities of the elder. Thus the potentially infinite computation produced by the non-checker processes is "$s_0^{(k)}, s_1^{(k)}, \cdots (k = 1, 2, 3, 4)$", where the local states indicate locations or activities.

Non-checker processes send *control messages* among each other to establish the *happen-before* relation. They also send *checking messages* to P_{che} which collects local states with logical timestamps from the non-checker processes and then constructs the lattice at runtime [13], [16]. The lattice serves as the basis for the online checking of specified properties.

B. Formal Specification of the Temporal Property

To ensure that the elder never goes to bed before taking medicine, we should always guarantee that "the elder should not go to sleep until the medicine is taken". Thus, $\Phi_{elder} = \forall(\neg\text{sleep} \cup \text{medicine})$ should be evaluated to be "true" on each state.

In the scenario, we decompose atomic propositions defined on the CGSs into local predicates which can be detected on non-checker processes. Specifically, the application is concerned of the following four local predicates:

- $LP^{(1)}$ = "$Reader^{(1)}_{Bed}$ *detects the elder*"
- $LP^{(2)}$ = "$Reader^{(2)}_{Med}$ *detects the elder*"
- $LP^{(3)}$ = *"Action of the elder is lying down"*
- $LP^{(4)}$ = *"Action of the elder is pouring water"*

Based on the local predicates, we can define atomic propositions $sleep$ and $medicine$ over CGSs:

- $sleep = LP^{(1)} \wedge LP^{(3)}$
- $medicine = LP^{(2)} \wedge LP^{(4)}$

Note that this example further illustrates the need for adopting 3-valued semantics. Intuitively, the application should raise an alarm only if the elder went to *sleep* before *medicine* was taken. The application should eventually turn itself off, when *medicine* was taken before *sleep* occurred, and should return '?' otherwise meaning that further observations are needed.

VI. Performance Measurements

In this section, we conduct experiments to demonstrate the benefits of specifying contextual properties in CTL$_3$, as well as the cost-effectiveness of the CTL$_3$ checking algorithm, in the elderly care scenario. The algorithm has been implemented in one of our research projects - *Middleware Infrastructure for Predicate detection in Asynchronous environments* (MIPA) [16], [23]. We first describe the experiment setup and then discuss the evaluation results.

A. Experiment Setup

We simulate the scenario discussed in the case study. Specifically, sensors collect context data per minute. Duration of local contextual activities on non-checker processes follows the Poisson distribution. The average duration of the contextual activity is 25 minutes and the interval between contextual activities is 5 minutes.

In the experiments, we mainly study whether the application is frequently encountered with the case of being inconclusive. We define the *Probability of Occurrences of the Third Value "inconclusive"* $Prob_{inc}$. $Prob_{inc}$ is calculated as the ratio of $\frac{NCheck_{inc}}{NCheck_{Lattice}}$, where $NCheck_{Lattice}$ denotes the total number of checkings of contextual properties, and $NCheck_{inc}$ denotes how many times the application is encountered with the case of being inconclusive.

In addition, we study performance of the online checking algorithm for CTL$_3$. The *Response Time* ($time$ between one checking is triggered and the checking is finished) and *Space Cost* (*space* for storage of the lattice, regarding one node in lattice as one unit of storage) are investigated.

B. Necessity of 3-valued Semantics and Performance of CTL$_3$ Checking

In this experiment, we study the necessity of 3-valued semantics. Specifically, we conduct the experiments 20 times with each consisting of 100 times of property checkings. We record the number of checkings which are involved with the third value *"inconclusive"* during each experiment.

As shown in Fig. 3, on average, checkings involved with the third value *"inconclusive"* makes up more than a third of all the checkings. The situation in which the third value is used means that the application will take wrong adaptations if traditional boolean semantics is adopted. The experiment shows that in PvC scenarios, we are often encountered with the case of being inconclusive, which justifies the adoption of 3-valued semantics.

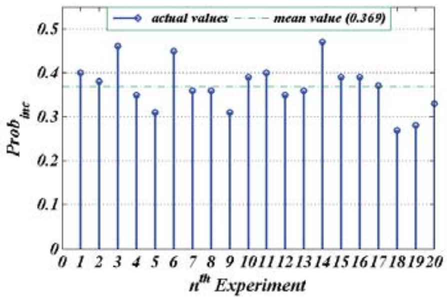

Figure 3. The third value "inconclusive" occurs frequently.

To study the performance of the algorithm, we keep a successive record of 1000 times of checkings, take average on *space* and *time* cost for every 50 times, and then study the relation between *space* and *time*. As shown in Fig. 4, *time* increases almost linearly in *space*. This guarantees that the CTL$_3$ checking algorithm can be efficiently applied in PvC scenarios. Moreover, we can manipulate the lattice and reduce the space cost at runtime. Thus, the CTL$_3$ checking algorithm can be integrated with online manipulation of the lattice to further reduce the *space* and *time* cost.

VII. Related Work

Our work can be posed against two areas of related work: context-aware computing and detection of global predicates over asynchronous computations.

In the area of context-aware computing, many schemes have been proposed for detection of contextual properties [3]. Detection of the temporal properties is of great importance. However, it is challenged by the intrinsic asynchrony in PvC environments [11], [12], [13]. Existing schemes mainly focus on how to cope with the asynchrony. It is not

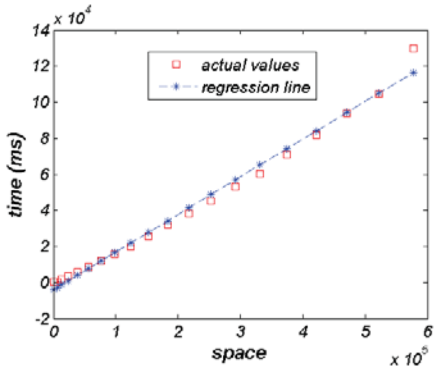

Figure 4. Space (taking one single CGS as one unit) vs. time

sufficiently discussed how *formal specification* of contextual properties is affected by the asynchrony. We argue that the formal specification for contextual properties should adopt the notion of branching time, to capture the intrinsic uncertainty resulting from the asynchrony.

In detection of global predicates over asynchronous computations, different types of predicates are proposed to describe specific global state of the distributed system [24], [25]. In [15], a unified predicate detection framework based on CTL was proposed to describe temporal evolution of global states of a distributed system. Classical CTL is defined over infinite traces of possible system state evolution. However, in context-aware computing scenarios, the application is usually faced with finite prefix of the (potentially infinite) trace of the PvC environment state evolution. Meanwhile, formal specification of temporal properties should also provide convenient support to cover the case of being inconclusive when only finite observations of global state evolution are obtained.

In theoretical work on temporal logic, 3-valued semantics is not novel [26], [27]. [26] defined their 3-valued temporal logic based on *partial Kripke structure* whose atomic propositions can have a third truth value *"unknown"*. However, our 3-valued semantics arises from the *finite* observation of system which is regarded as a *traditional transition system* (or Kripke structure). Moreover, we present an online checking algorithm for CTL$_3$ which is quite different from that in [26]. In addition, we focus on the challenges posed by PvC scenarios, while [26], [27] had investigated the expressiveness of 3-valued models in the context of automatic abstraction and model checking.

VIII. Conclusion and Future Work

In this work, we study how to enable formal specification and runtime detection of temporal contextual properties in asynchronous PvC environments. The CTL$_3$ scheme is proposed which i) inherits the notion of branching time from classical CTL; ii) adopts 3-valued semantics to delineate the satisfaction of temporal properties over finite traces of environment state evolution.

In our future work, we need to improve the scalability of our proposed scheme. Specifically, the checking algorithm for CTL_3 should be implemented more efficiently via delicate data structures. The space cost can be reduced by combining our algorithm with other techniques such as sliding widow. From a practical point of view, various tools still need to be implemented, in order to apply the CTL_3 scheme in real context-aware computing environments. Moreover, more comprehensive and realistic experimental evaluations are also necessary.

ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China (No. 60903024, 61021062), the National 973 Program of China (2009CB320702).

REFERENCES

[1] D. Garlan, D. Siewiorek, A. Smailagic, and P. Steenkiste, "Project aura: Toward distraction-free pervasive computing," *IEEE Pervasive Computing*, vol. 1, no. 2, pp. 22–31, 2002.

[2] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, 2002.

[3] C. Xu, S. C. Cheung, W. K. Chan, and C. Ye, "Partial constraint checking for context consistency in pervasive computing," *ACM Trans. Softw. Eng. Methodol.*, vol. 19, no. 3, pp. 1–61, 2010.

[4] A. Dey, "Providing architectural support for building context-aware applications," Ph.D. dissertation, Georgia Institute of Technology, 2000.

[5] K. Henricksen and J. Indulska, "Developing context-aware pervasive computing applications: Models and approach," *Pervasive Mob. Comput.*, vol. 2, no. 1, pp. 37–64, 2006.

[6] A. Ranganathan and R. H. Campbell, "An infrastructure for context-awareness based on first order logic," *Personal Ubiquitous Comput.*, vol. 7, pp. 353–364, Dec 2003.

[7] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, pp. 8–23, Sep 1990.

[8] A. Pnueli, "The temporal semantics of concurrent programs," *Theoretical Computer Science*, vol. 13, no. 1, pp. 45–60, 1981.

[9] M. Ben-Ari, Z. Manna, and A. Pnueli, "The temporal logic of branching time," in *Proc. of the 8th ACM SIGPLAN-SIGACT symposium on Principles of Program. Lang.*, 1981, pp. 164–176.

[10] A. Bauer, M. Leucker, and C. Schallhart, "Runtime verification for ltl and tltl," *ACM Trans. on Softw. Eng. and Methodol. (TOSEM)*, vol. 20, no. 4, p. 14, 2011.

[11] 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 Distrib. Syst.*, accepted in May 2011.

[12] Y. Huang, X. Ma, J. Cao, X. Tao, and J. Lu, "Concurrent event detection for asynchronous consistency checking of pervasive context," in *Proc. IEEE Intl. Conf. on Pervasive Computing and Communications (PERCOM'09)*, Mar 2009.

[13] T. Hua, Y. Huang, J. Cao, and X. Tao, "A lattice-theoretic approach to runtime property detection for pervasive context," in *Proc. of the 7th Intl. Conf. on Ubiquitous Intelligence and Computing*, 2010, pp. 307–321.

[14] R. Schwarz and F. Mattern, "Detecting causal relationships in distributed computations: in search of the holy grail," *Distrib. Comput.*, vol. 7, no. 3, pp. 149–174, 1994.

[15] A. Sen and V. Garg, "Formal verification of simulation traces using computation slicing," *IEEE Trans. on Comput.*, pp. 511–527, 2007.

[16] MIPA - Middleware Infrastructure for Predicate detection in Asynchronous environments. http://mipa.googlecode.com.

[17] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *CACM*, vol. 21, no. 7, pp. 558–565, 1978.

[18] F. Mattern, "Virtual time and global states of distributed systems," in *Proc. Intl. Workshop on Parallel and Distrib. Algorithms*, 1989, pp. 215–226.

[19] O. Babaoğlu and K. Marzullo, "Consistent global states of distributed systems: Fundamental concepts and mechanisms," 1993.

[20] F. Chen and G. Roşu, "Parametric and Sliced Causality," in *Proc. of the 19th Intl. Conf. on Computer Aided Verification (CAV'07)*, vol. 4590, 2007, pp. 240–253.

[21] C. Baier and J. Katoen, *Principles of model checking*. The MIT Press, 2008.

[22] S. Kleene, *Introduction to metamathematics*. Wolters-Noordhoff, 1971.

[23] J. Yu, Y. Huang, J. Cao, and X. Tao, "Middleware support for context-awareness in asynchronous pervasive computing environments," in *IEEE/IFIP Intl. Conf. on Embedded and Ubiquitous Computing (EUC'10)*, Dec 2010.

[24] R. Cooper and K. Marzullo, "Consistent detection of global predicates," in *Proc. ACM/ONR Workshop on Parallel and Distributed Debugging*, 1991, pp. 167–174.

[25] V. K. Garg and B. Waldecker, "Detection of strong unstable predicates in distributed programs," *IEEE Trans. on Parallel and Distrib. Syst.*, vol. 7, pp. 1323–1333, Dec 1996.

[26] G. Bruns and P. Godefroid, "Model checking partial state spaces with 3-valued temporal logics," in *Computer Aided Verification: 11th Intl. Conf., CAV'99*, no. 1633, 1999, p. 274.

[27] P. Godefroid and R. Jagadeesan, "On the expressiveness of 3-valued models," in *Verification, Model checking, and Abstract Interpretation: 4th Intl. Conf.*, vol. 4, 2003, p. 206.