Formal analysis of pervasive computing systems

Yan LIU

Xian ZHANG

Jin Song DONG

Yang LIU

Jun SUN
Singapore Management University, junsun@smu.edu.sg

See next page for additional authors

Formal Analysis of Pervasive Computing Systems

Yan Liu, Xian Zhang, Jin Song Dong
School of Computing, National University of Singapore
Email: {yanliu, zhangxi5, dongjs}@comp.nus.edu.sg

Yang Liu
Temasek Lab, National University of Singapore
Email: tslliuya@nus.edu.sg

Jun Sun
Singapore University of Technology and Design
Email: sunjun@sutd.edu.sg

Jit Biswas
Institute for Infocomm Research
Email: biswas@i2r.a-star.edu.sg

Mounir Mokhtari
CNRS-IPAL/Institut TELECOM
Email: Mounir.Mokhtari@it-sudparis.eu

***Abstract*—Pervasive computing systems are heterogenous and complex as they usually involve human activities, various sensors and actuators as well as middleware for system controlling. Therefore, analyzing such systems is highly non-trivial. In this work, we propose to use formal methods for analyzing pervasive computing systems. Firstly, a formal modeling framework is proposed to cover the main characteristics of pervasive computing systems (e.g., context-awareness, concurrent communications, layered architectures). Secondly, we identify the safety requirements (e.g., free of deadlocks and conflicts etc.) and propose their specifications as safety and liveness properties. Finally, we demonstrate our ideas using a case study of a smart nursing home system. Experimental results show the effectiveness of our approach in exploring system behaviors and revealing system design flaws such as information inconsistency and conflicting reminder services.**

***Keywords*-Pervasive Computing, Formal Modeling, System Verification**

I. Introduction

Pervasive computing systems are context-aware and adaptable to the evolving environments [1]. The changes in the environment are monitored and recorded in the system as contexts. If a particular event happens, the system is able to adapt itself to the changes. In the current literature, these systems usually adopt a layered design with sensors in the hardware layer to acquire environment contexts; inference engines in the middleware layer to manage and reason these contexts as well as make adaptation decisions; services in the application layer to invoke actuators to execute the decisions. Consequently, the heterogeneity of technology and massive ad hoc interactions among layers make pervasive computing systems highly complicated [2]. Furthermore, various environment inputs and unpredictable user behaviors cause the system behaviors beyond control, especially when multiple users are interacting with the system simultaneously. Therefore, it is a challenging task to guarantee the correctness and even the safety of such systems. Traditional validation methods such as simulation and testing have their limitations in performing this task, i.e., these methods can only cover partial system behaviors based on the selected scenarios.

In this work, we propose to use formal methods to analyze pervasive computing systems to overcome these limitations. Our contributions are three-folds as explained below.

Firstly, we propose a framework to formally model the system design and the environment inputs. Important characteristics of pervasive computing systems such as context-awareness, layered architecture and concurrent communications are discussed. Modeling patterns for these features are provided and illustrated with examples. We adopt CSP# [3] as the sample modeling language for its rich set of syntax in modeling concurrent system with hierarchies. Dong et al. [4] and Coronato et al. [5] proposed to model such systems using TCOZ [6] and Ambient Calculus [7] respectively. Although these languages are good at modeling the communications and mobility features respectively, the support for modeling hierarchical structures is limited. Most importantly, there is very little tool support for these languages, which limits the usage and applicability of their approaches.

Secondly, we identify critical properties of pervasive computing systems and provide their specification patterns in corresponding logics. According to the stakeholders (designers, engineers and users of these systems), safety requirements are essential to pervasive computing systems. Arapinis et. al. in [8] proposed some critical requirements of a homecare system. For instance, “Sensors are never offline when a patient is in danger” or “If a patient is in danger, assistance should arrive within a given time”. In our work, we classify the important requirements into safety properties (nothing *bad* happens) and liveness properties (something *good* eventually happens). Furthermore, formal specification patterns of these properties are proposed. As a result, we can verify the critical properties against the system model by using automatic verification techniques like model checking [9]. Hence, design flaws can be detected at the early design stage.

Thirdly, we demonstrate a case study of a smart healthcare system for mild dementia patients, AMUPADH [10]. A typical workflow of this formal analysis process is shown in Figure 1. We start the project with collecting requirements through multiple visits to the nursing home and interviews

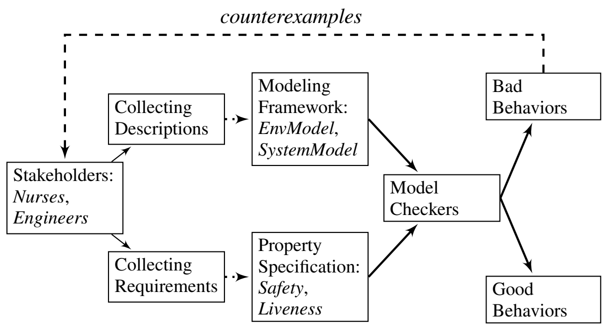

counterexamples
Stakeholders: Nurses, Engineers
Collecting Descriptions
Collecting Requirements
Modeling Framework: EnvModel, SystemModel
Property Specification: Safety, Liveness
Model Checkers
Bad Behaviors
Good Behaviors

Figure 1: Formal Analysis Workflow

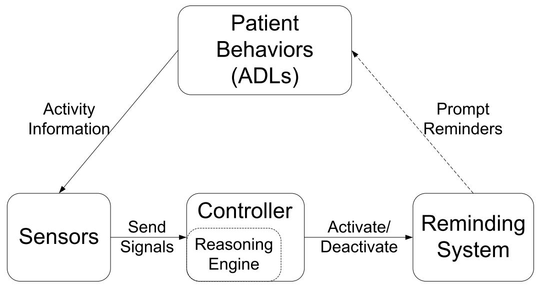


Figure 2: An Overview of the Smart Bedroom System

of nurses/doctors. From discussions with system designers, we learn that AMUPADH is a typical pervasive computing system which incorporates sensors and a reasoning engine to understand the patients' intentions and provides reminder services to help them. Additionally, AMUPADH has a multi-person sharing environment which exhibits additional complexity in terms of concurrent interactions. Then, we model the user behaviors and system design based on our modeling framework using CSP# language. Critical properties such as deadlock freeness, guaranteed reminder service and conflicting reminders tests are verified using PAT model checker [11] (a self-contained framework for modeling, simulating and reasoning of concurrent and real-time systems). Multiple unexpected bugs such as information inconsistency are exposed.

In the rest of the paper, we introduce AMUPADH system in Section II. The modeling framework and critical properties of pervasive computing systems are demonstrated in Section III and IV respectively. Section V illustrates the case study and reports the unexpected bugs we found. Related works are discussed in Section VI. Lastly, Section VII concludes the paper with future work.

II. A Motivating Example: AMUPADH - An Ambient Assisted Living System for Dementia Healthcare

Dementia is a progressive, disabling, chronic disease common in elderly people. Elders with dementia often have declining short-term memory and have difficulties in remembering necessary activities of daily living (ADLs). However, they are able to live independently or in assisted living facilities with little supervision.

Ambient Assisted Living systems equip the environment with a spectrum of computation and communication devices that seamlessly augment human thoughts and activities. The system developed in AMUPADH is able to monitor the patients' behaviors using activity recognition techniques (sensors and reasoning rules) and offer help to the patients (prompt reminders through actuators such as speakers etc.).

A. System Overview

The architecture of the system is shown in Figure 2. The system is deployed in a bedroom with two beds and a shower facility. Different kinds of sensors are deployed in the room to capture environment changes. For instance, the pressure sensor under a mattress is used to detect whether the bed is empty or occupied. Sensors communicate with the *controller* via wireless network. The *controller* in the middleware interprets sensor signals into low-level contexts from which high-level contexts are inferred by the *reasoning engine*. This reasoning task is performed based on a set of predefined rules written in Drools[1] (based on First Order Logic). Evaluation of these rules is triggered by a sensor message or periodically by a timer. In the case that a rule is satisfied, the system will adapt to a new state by updating internal variables or invoking reminder services. For example, if the activity of patient sleeping on a wrong bed is recognized, the system will prompt a reminder requesting him to use his own bed.

B. Sensors

In AMUPADH, four types of sensors are deployed in the bedroom and shower room to monitor the activity of dementia patients as shown in Figure 3.

- **RFID Reader** is for identification and tracking. There are two readers placed beside the doors to detect who has entered the rooms respectively and two attached to each bed to identify who is using the bed. Each patient is wearing an RFID tag placed in a wrist band.
- **Pressure Sensor** is placed under the mattress of each bed to detect activities in bed, e.g., sitting or lying.
- **Shake Sensor** can detect vibration. They are attached to water pipe and soap dispenser for sensing the usage of water tap and soap respectively.
- **Motion Sensor** (A.K.A. passive infrared sensor (PIR)) can measure infrared light radiating from objects in its range. It is used to detect the presence of the patient in the shower room.

[1]Drools Expert: http://www.jboss.org/drools/drools-expert.html

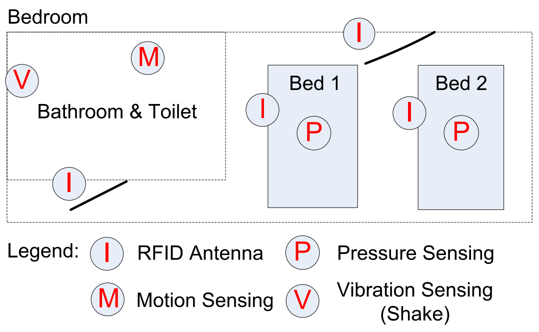


Figure 3: Sensor Layout in the Bedroom

C. Controller

In the *Controller*, contexts are managed and inferenced. It has two components i.e., the *Main Interface* interprets the sensor signals and triggers the evaluation of all rules when a sensor message arrives; the *Context Checker* evaluates all rules every 5 minutes. The rules are written in Drools and evaluated by the business rule engine, Drools Expert. They are specified with a name, a condition formed of predicates and the adaptation actions. For example, the rule for detecting sitting bed for too long is specified as follows.

```
rule "personA sat on Bed A for too long (30mins)"
  when
    Sensor( id == "pressureBedA",
          pressureState == Sensor.pressure_state.SITTING,
          duration > 30 )
    $x : XMPPInterface()
  then
    $x.SendData("ACTIVITY.error."+"SitBedTooLong"+"."
         +"personA");
end
```

The condition of this rule consists of three context variables: the sensor's *id*, status and timer. This rule can be interpreted as: the message *ACTIVITY.error.SitBedTooLong.personA* will be delivered to the reminding system if the *SITTING* status of pressure sensor on bed A has lasted for more than 30 minutes. The messages are sent out via a shared bus. The full set of 23 rules used in the system is listed in [12].

D. Reminding System

The reminding system in the application layer activates/deactivates reminders based on the incoming messages from *Controller*. For example, if the message is *ACTIVITY.error.SitBedTooLong.personA*, the reminding system decodes it and knows patient A (named Jim) has sleeping problems. Thus it invokes a speaker and prompts *'Jim, you have been sitting on bed for a long time, please go to sleep'*. This reminder will be continuously repeated until proper actions have been taken. If the prompts reaches the maximum number, this reminder will be sent to nurses.

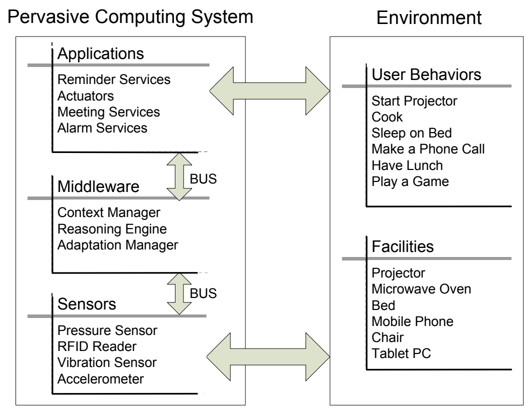


Figure 4: Architectures of Pervasive Computing Systems

III. A Modeling Framework for Pervasive Computing Systems

Pervasive computing systems are carefully designed for users who expect the system to aid in their daily life. They are usually complex and adopt a layered architecture as shown in Figure 4. In this section, we discuss the important features of pervasive computing systems layer-by-layer and propose corresponding modeling patterns for them. Besides, environment inputs perform an important role in pervasive computing systems. Thus, along with the modeling of system components, we also propose modeling patterns for different environment aspects which are usually not included in most complex systems models.

A. Modeling Environments

Pervasive computing systems seamlessly interact with the environments and acquire context inputs from the users and objects like TVs and Beds. To some extent, pervasive computing systems are driven by the environment context change (we call it *scenario* here). For example, a person entering a room which is previously empty will trigger the lights to be switched on; or when the system detects the time is 9:00pm, a take-medicine-reminder will be sent to the patient. Thus, it is important to model the scenarios with the system design. Meanwhile, the scenario model is also important for generating meaningful counterexamples so as to alleviate the burden of analyzing verification results.

Modeling Activities and Environment Objects: User behaviors are various and usually unpredictable. For most pervasive computing systems, we can observe that: 1) the system usually targets a certain group of activities and ignores other irrelevant ones; 2) relevant user activities are determined but the order of them is unpredictable. For instance, a person enters the bedroom, then he may directly go to sleep or he could possibly enter the shower room for other activities. In practice, targeted activities can be provided by system designers. We use a shower room scenario to demonstrate the modeling patterns.

In the shower room, a user performs many activities such as wandering or turning on the shower tap. These activities can be modeled as *events* which are abstractions of the observations. For example, an activity represented as event *exitShowerRoom* is an observation of the user's behavior of leaving the shower room. However, it requires more advanced language constructs such as non-deterministic choices to model all possible orders of activities. We explain the idea using a CSP# model of the shower room scenario. All the possible activities the patient can do in the room are modeled as different choices and they are enclosed into a process named *PatientShowerRoom*.

```
PatientShowerRoom() = exitShowerRoom → PatientOutside()
            □ turnOnTap → PatientShowerRoom()
            □ turnOffTap →  PatientShowerRoom()
            □ wandering →  PatientShowerRoom()
            □ useSoap → PatientShowerRoom();
```

Here, the operator □ represents the non-deterministic choice. It operates this way that the process *PatientShowerRoom* randomly choose an activity such as *turnOnTap* to execute. Then it may transfer control to itself again and choose *useSoap* to execute. It is guaranteed that all possible orders of activities are generated using state space exploration techniques like model checking.

However, there might exist some unrealistic orders of events. For example, there is a sequence which contains two consecutive events of *turnOnTap*. Obviously, the patient cannot perform turning tap on activity again if the tap is turned on already. In order to eliminate such cases, we need to model these constraints such that the patient's behavior is synchronized with the status of the object being used. In fact, it is essentially the problem of modeling synchronous behaviors. We propose to use event synchronization in CSP# and give an example of shower tap model in the following. Other solutions are possible such as using a global variable or synchronous channels.

```
ShowerTap() = turnOnTap → turnOffTap → ShowerTap();
Env() = PatientShowerRoom() ∥ ShowerTap();
```

The constraint of using tap behaviors is modeled as if *turnOnTap* event happens, it will be disabled until the *turnOffTap* activity is performed. The two processes *PatientShowerRoom* and *ShowerTap* are composed to be a complete model of the environment, *Env*. Here, the operator ∥ denotes parallel composition. Its operational semantic says that the executions of the composed processes must be synchronized on common events appearing in all of them. Interested readers can refer to [3] for more details. Here, the *turnOnTap* event becomes a common event between the two processes.

Modeling Location Transitions: While modeling the patients behaviors, we divide the activities according to the locations where they can be performed. In the *PatientShowerRoom* model, if the event *exitShowerRoom* is engaged, the process will pass control to the *PatientOutside* process. Thus, only activities outside can be selected to run while activities in the shower room are disabled. This modeling approach is to reflect the location transitions in the model and to generate realistic sequences of activities.

Modeling Multiple Users: In multiple-user sharing environment, the activities that different users can perform in a certain location are usually the same. However, in some cases, these activities need to be differentiated. For example, in AMUPADH, the system tracks different patients using RFID tags. Thus, the sitting on bed behavior performed by patient1 and patient2 are different from the system's point of view. We model this requirement using the process parameters and events with indexes. In the following, we provide the behavior model of the patient using bed where identify information is important.

```
PatientBed(i) = sitOnBed.i  → PatientBed(i)
           □ lieOnBed.i  → PatientBed(i)
           □ leaveBed.i  → PatientBed(i);
```

Parameter i in process *PatientBed*(i) represents the identity of the patients. This identity variable is also attached to events so as to differentiate the activities performed by different patients.

B. Modeling System Design

Pervasive computing systems share the features such as layered architecture and concurrent communications. A common architecture of such systems is shown in Figure 4. In the following, we discuss these common features and their modeling layer by layer.

1) Modeling Sensor Layer: There are a lot of interesting problems in this layer. First of all, there are different communication patterns like synchronous communication or asynchronous message passing. These communications form the basic functionality of sensors. Additionally, different sensors have different frequencies of sending messages. For example, RFID reader sends a signal to system every 1 second while pressure sensor sends every 10 seconds. This issue may cause the system to make wrong adaptations since the information of the environment may not be completely refreshed at some time point. Finally, sensors have limited power supply and may fail from time to time. These two problems regarding the different sending rates and unstable working conditions of sensors create many uncertainties in pervasive computing systems.

Nonetheless, problems might also exist in the wireless network such as message loss. We skip this part since research of model checking wireless networks has been done extensively in the literature [13]. The details about signal encoding/decoding and message transmission via wireless networks are abstracted away for simplicity in our work.

Modeling Concurrent Interactions: Sensors interact with the environment by detecting events and report sensed contexts by transmitting signals to middleware. The behaviors of detecting and transmitting can be abstracted to two modeling patterns which are synchronous events and message passings respectively. Event synchronization has been introduced in Section III-A. As for message passing, there are different modeling patterns in different languages. Some languages support synchronous channels through which the sending and receiving events are synchronized. In other languages, broadcast channels or asynchronous channels with buffers are supported. In the following, we model the shake sensor using a synchronous channel.

```
channel port 0;
Shake_Sensor() = (
        turnOnTap  → port!Shake.UnStationary  → Skip
      □ turnOffTap  → port!Shake.Stationary  → Skip
      ); Shake_Sensor();
```

Here, *port* is the synchronous channel defined for the shake sensor to communicate with middleware. *Shake*, *UnStationary* and *Stationary* are integer constants representing the sensor's ID and possible statuses. In the model, the shake sensor sends out the signal *UnStationary* when the tap is turned on. Note that CSP# supports multi-process synchronization that the event *turnOnTap* can be synchronized in all three processes.

Modeling Frequency: Sensors are tuned to have different sending rates due to their functionalities and the purpose of saving energy. However, if the rates are not carefully calculated, the system may work incorrectly. To analyze these behaviors, we propose to use timed modeling languages such as Stateful Timed CSP (STCSP) [14] or Timed Automata (TA) [15]. The modeling pattern of sending rates using STCSP would be as follows.

```
FSR_Sensor() = (
       sitOnBed  ↠ port!FSR.Sitting ↠ Skip
     □ lieOnBed ↠ port!FSR.Lying ↠ Skip
     □ leaveBed ↠ port!FSR.Empty ↠ Skip
     □ nothing ↠ port!FSR.Empty ↠ Skip
     ); Wait[10]; FSR_Sensor();
```

Here, operator ↠ denotes the urgent event in its left hand side which cannot be interleaved by other timed events. *Wait*[*t*] is the syntax to model the process idling for *t* time units. The above process models the periodic behaviors of the pressure sensor which senses the environment for certain activities and immediately transmits its status. Then it idles for 10 time units and starts sensing again.

Modeling Sensor Failures: Sensors have limited accuracy, so that they may fail to detect certain events. They could also run out of battery and then fail to send the signals. Intuitively, we model this with probabilistic modeling languages such as Probabilistic CSP (PCSP) [16] or Probabilistic Timed Automata (PTA) [17].

```
RFID_Reader() =
       enterBedroom.1  → port!RFID.PersonA  → Skip
     □ enterBedroom.2  → port!RFID.PersonB  → Skip;

MalSensor() = pcase{ 9: RFID_Reader()
                     1: fail  → Skip }; MalSensor();
```

Here, *pcase* is a syntax for modeling probabilities. 9 and 1 are probability weights here. This process models that the RFID reader works correctly with probability of 90%.

In summary, different issues in the sensor layer can be modeled using different language constructs. Notice that the two modeling languages (i.e., STCSP, PCSP) we adopted are both extensions of CSP# language. As demonstrated in above examples, our intention is that it is easy to start with a simple model and extend it with richer features with minimum efforts.

2) Modeling Middleware Layer: As shown in Figure 4, middleware performs the tasks of managing and reasoning contexts as well as making adaptation decisions. Messages received from sensors will trigger an update of the system knowledge/contexts. The status of a sensor is one kind of contexts. Context variables are modeled using shared variables in supporting modeling languages.

Furthermore, the reasoning engine performs reasoning by evaluating predefined rules whose conditions are propositions of context variables. A common practice for specifying rules is to use guarded processes or if-else statements. The following example models the rule in Section II-C in CSP#:

```
Rule() = if(sensors[Pressure_Sensor] == SITTING &&
            Duration[Pressure_Sensor] > 30)
            res!Act.SitTooLong.1 → Skip;
```

Finally, an adaptation decision will be made based on the reasoning results and sent to the application layer to execute. This again can be modeled by message passing patterns. For the above example, if the *rule* which interprets that someone is sitting on bed for more than 30 time units, a message will be sent to the application layer through the channel *res*.

3) Modeling Application Layer: Application layers vary according to different implementations. However, we may only care about the responsive actions which will affect the end users. Thus we focus on modeling of how the adaptation decisions are executed. For instance, in the AMUPADH system, the reminding system is modeled as follows:

```
Reminder() = res?status.rid.pid → (
              [status == Act]ActivateReminder(rid,pid)
             □[status == Deact]DeactReminder(rid,pid)
             ); Reminder();
ActivateReminder(rid,pid) =
             updatereminder[rid][pid] = true → Skip;
```

By decoding the message received from middleware, the workflow of reminder system diverts according to the *status* command. If it is an *Act* command, the system activates reminder *rid* to patient *pid* by calling *ActivateReminder*(*rid*, *pid*) process. Similar logic applies for deactivating a reminder.

C. Compose a Complete Model

In pervasive computing systems, different components in different layers cooperate to fulfill the system goals. However, how to model this cooperate relations are left to be discussed till now. From a careful study, we discover that there are three kinds of relationships between these components which are sequential, independent and concurrent relations. Sequential relation means the execution of the components is strictly sequential according to the workflows of the system. Components that are completely unrelated to each other execute independently. As for concurrently related components, they have synchronized behaviors. These relations can be well supported in hierarchical languages such as CSP#. Respectively, these three relations can be modeled as sequential, interleave and parallel compositions using operators ; , ||| and ‖ respectively. Examples here may reuse some process names in above models. Note that parallel composition has been introduced in modeling activities in the environment.

```
Sensors() = Shake_Sensor() ||| FSR_Sensor();
Middleware() = ContextManager(); ReasoningEngine();
               AdaptationManager();
```

Here, since each sensor in the environment works independently, the sensor layer model *Sensors*() is composed by the interleave operator. On the other hand, in the middleware layer, the three components are executed sequentially as determined in the workflow. Therefore, the middleware model *Middleware*() is composed using sequential operator.

IV. Properties of Pervasive Computing Systems

After system engineers finished the design of a pervasive computing system, they are often asked to provide guarantees for correctness and even safety requirements. They may be asked to answer general questions like "Is the system free of conflict adaptations?" or "Will the services deliver when they are supposed to?". These high level requirements cannot be validated against the system thoroughly using traditional techniques like testing. However, they can be specified and verified using formal methods. For example, using model checking technique, the first question can be verified in the following steps. First, define the conflict adaption scenario as a state; Secondly, use reachability verification algorithms to exhaustively search the system state space to see if such a state is reachable. In this section, we discuss the critical properties and propose their specification patterns.

A. Desirable Properties

Properties regarding the good behaviors of the systems are desirable.

1) Deadlock freeness: Deadlock freeness is one of the important safety requirements. Deadlock is a situation that the system reaches a state where no more actions can be performed. It can lead to serious consequences such as falling of the patient is not being alerted to a nurse. Deadlock checking is supported in most model checking tools.

2) Guaranteed Services: Well designed application services determine fundamental responsive behaviors of pervasive computing systems. For example, in a smart meeting room, upon detection of some one entered the room, a

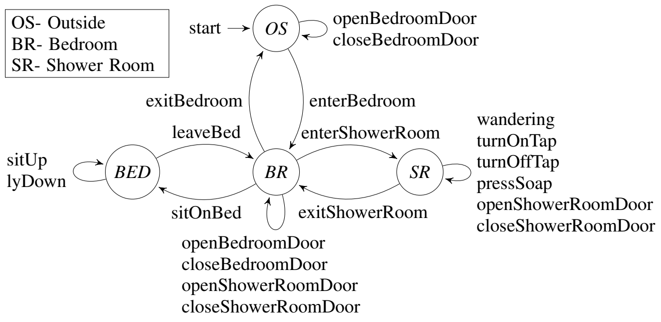


Figure 5: Patient Behaviors

service will be scheduled to run that it will invoke an actuator to automatically turn on the lights. Effectiveness of these services is an important measurement of the system for the sake of users. To specify this requirement, we propose patterns of liveness properties using Linear Temporal Logic (LTL). For example,

```
□(PatientWandering → ◇ LeaveRoomReminder)
```

Here, $\Box$ and $\Diamond$ are operators in LTL which read "always" and "eventually". This formula specifies the property meaning "Always when *PatinetWandering* situation happens, the service *LeaveRoomReminder* will be eventually delivered".

The services are usually required to be delivered in bounded time. Obviously, it is certainly undesirable if the reminder is sent too late that even the patient has left the room. To specify the bounded liveness properties, one can use Timed Computational Tree Logic (TCTL) which extends CTL with clock constraints. The other possible solution is to bound the target system model with *deadline* semantics in some real time modeling languages such as STCSP.

3) Security: Since pervasive computing systems carry lots of environment information including the user's confidential profiles, it is critical to protect privacy. Leakage of information can compromise the safety of the user and his or her belongings. For instance, food delivery person should not have access to the patients medical profile. Properties to describe security problem can be specified in many kinds of logics such as LTL. For example,

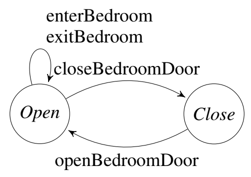
```
□(FoodDeliveryPerson → not (◇ AccessPatientProfile))
```

Model checking techniques for security problems are proposed in papers such as [18].

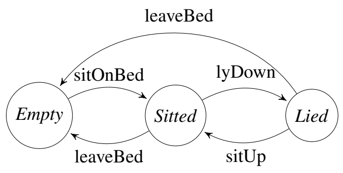
B. Testing Purposes

To test the system after being deployed is cumbersome considering the reengineering workload. Fortunately, those unwanted scenarios can be specified in properties and checked using reachability verification algorithms.

1) System Inconsistency: Failures of sensors and wireless networks may cause contexts of the environment in the

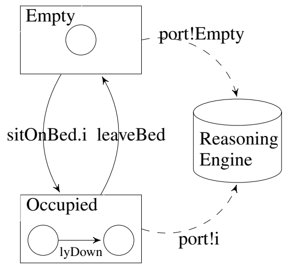


(a) Bedroom Door Behavior

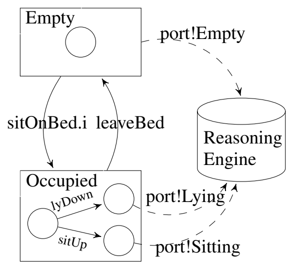


(b) Bed Behavior

Figure 6: Surrounding Environment



(a) Bed RFID Reader



(b) Bed Pressure Sensor

Figure 7: Sensor Behaviors

system to be out of date. Thus system knowledge can be inconsistent with actual environments. By defining such conflicting states, you can test again the system model to see if such a state is reachable.

2) Conflicting/ False Services: To guarantee the services being eventually delivered is not enough. It is also important to check if these services are sent properly. Some problems have been reported by domain experts such as conflicts of reminders [19]. These problems are especially common in multi-user systems. For example, in AMUPADH, two conflicting reminders are prompted at the same time that one asks the patient to leave shower room while the other asks the patient to use soap to continue showering. This causes the confusion of the patient and could agitate them. Another scenario is that the reminder is sent to the wrong person. These problems can be specified in reachability properties.

3) Properties in rules: Rule-based reasoning engines are popular in pervasive computing systems. The correctness of rules is essential to the correct behaviors of systems. Problems of these rules include duplicated rules, conflict rules and unreachable rules. This is also easy to specify. For example, to check whether a rule is unreachable, the condition of the rule can be defined as a state and property can be expressed as testing if the state is reachable.

V. Case Study: Formal Analysis of AMUPADH

The proposed approach is applied to analyze AMUPADH. We adopt CSP# modeling language since it supports most of the modeling patterns in the framework. Important properties are specified in reachability semantic and LTL formulae. PAT model checker is chosen to parse the model, build up the system state space and verify these properties. Experiment results are listed and unexpected bugs are reported.

A. System Modeling

In this section, we model the environments and the system design using our framework and use Labeled Transition Systems (LTS) for demonstration.

1) Environment Model: As shown in Figure 5 and 6. These LTSs can be generated using simulation function of PAT. In Figure 5, there are four possible locations that a patient can reside. The transition edges between states are labeled with patient's activities.

This patient model should be synchronized with objects within the surrounding environment. The objects that are modeled include doors of bedroom and washroom, beds and washroom taps. The behavior models of the doors and beds are shown in Figure 6a and 6b respectively.

2) Sensor Model: Different sensors are used in AMUPADH to monitor specific behaviors of the patients. For example, pressure sensors attached to the bed mattresses are for monitoring how the patients use the beds. The information captured by sensors is passed from sensors to the controller via a synchronized channel *port*. Every sensor possesses multiple unique states when made available to the system. Figure 7 shows the modeling of sensors using the bed RFID readers and bed pressure sensors as mentioned in Section II-B. Then, we combine all processes of sensors to one process *Sensors* using composition patterns.

```
Sensors()=Rfid_Bedroom()
          ||| (Rfid_Beds() ∥ FSR_Sensors())
          ||| (Rfid_ShowerRoom() ∥ PIR_ShowerRoom())
          ||| ShakeSensors();
```

3) Controller and Reasoning Engine Model: Inside the reasoning engine, rule evaluation is triggered by two processes, namely the *MainInterface* and *ContextChecker* processes. In order to model the periodical evaluation by *ContextChecker*, we use a constant integer *RATE* to represent the interval and *Duration* variable to record elapsed time. The *atomic* syntax used here is to ensure the process inside the block is executed without interference from other processes.

```
ReasonEngine()  = MainInterface() ||| ContextChecker();
MainInterface() =
      atomic{port?id.status → update{sensors[id]=status;
      Duration= call(setTimer,id,status,Duration)} →
      FireAllRules()};MainInterface();
ContextChecker()=
      atomic{update{Duration = call(tick,Duration,RATE)}
      → FireAllRules()};ContextChecker();
```

On receiving a message from any sensor, the *MainInterface* updates the sensor status and *Duration*. After that, the *FireAllRules* process is invoked to perform reasoning. In the model above, we use the syntax $call(setTimer, id, status, Duration)$ to call an external static function *setTimer* (written in C#) to update *Duration* according to the input of sensor *id* and *status*. This is a special feature in PAT, which allows users to separate

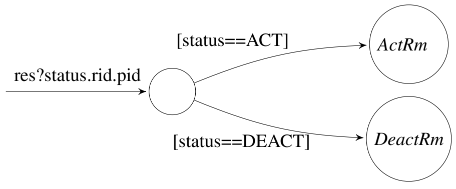


Figure 8: Reminding System Behaviors

complicated calculation from the high level model in order to have a simple model with efficient verification. The *ContextChecker* is similar to the *MainInterface* in updating sensor statuses and *Duration*, but does so in a periodic cycle instead of using a listener.

The process *FireAllRules* sequentially evaluates every rule independent of the results from previous cycles of rule evaluation and triggers proper actions such as setting a flag or sending a message to the reminding system. Messages are passed via a synchronous channel named *res*. We model every rule in a separate process. In the following, we list one rule to illustrate the modeling. The process *Rule*_14_1() models a complicated rule defined for recognizing the wandering behavior of the dementia patient. It says if the shake sensor on shower tap is stationary, the PIR sensor detects the patient's presence has lasted for 15 time units, the shower flag is still false and patient 1 is in the shower room, then patient1 is wandering in the shower room. Consequently, the reasoning engine sets the wander flag to true and passes a message to inform the reminding system that patient1 needs to be reminded to leave the room.

```
FireAllRules() = Rule0();
...
Rule_14_1() = if(sensors[ShakeTap] == STATIONARY &&
      sensors[PirShowerroom] == FIRING &&
      Duration[PirShowerroom] ≥ 15 &&
      !ShowerFlag && Location_Person[1] == SHOWERROOM){
      setFlag{WanderFlag = true} →
      res!Error.WanderingInShowerroom.1 → Rule_14_2()}
    else {Rule_14_2()};
...
```

4) Reminding System Model: In the system, reminders are activated/ deactivated upon receiving corresponding messages from the controller. As shown in Figure 8, the reminding system receives a triplet from the controller via channel *res*. This triplet consists of a command, behavior code and patient ID. If the command is *ACT*, the reminder *rid* will be activated and prompted to patient *pid*, otherwise the specified reminder will be stopped if it is active. The *ACT* and *DEACT* are command constants corresponding to *Normal* and *Error* in rule processes.

Finally we integrate all the sub-system models together into a process named *SmartRoom*() using composition patterns. Interested readers are referred to [12].

B. System Verification

In this section, requirements concerned by system designers and users (patients/ nurses/ doctors) are formally specified and verified.

1) Deadlock freeness (P1): Deadlock freeness property is directly supported in PAT using the keyword *deadlockfree*. However, the complete *SmartRoom*() model is too large to verify, we decompose it into subsystems according to locations. In fact the two subsystems shares only one context variable, the patient's location which can only be determined by one RFID tag (not shared). Thus, we argue that this decomposition fulfills the verification purpose.

```
P1.1 #assert SmartRoom() deadlockfree;
P1.2 #assert SmartBedroom() deadlockfree;
P1.3 #assert SmartShowerRoom() deadlockfree;
```

2) Guaranteed Reminders (P2): A well designed reminding service is very important for assisting elders with mild dementia. We list two reminder services in the bedroom and shower room scenarios respectively as follows. Other similar properties (P2.3-P2.6) are specified in [12].

Guaranteed Lying_Wrong_Bed Reminder (P2.1): This property states that when a patient is sleeping in a wrong bed, the system will always prompt the *LyingWrongBed* reminder eventually.

```
#define LyingWrongBed (sensors[RfidBed_1] ≠ EMPTY
          && sensors[RfidBed_1] ≠ 1);
#define RemindedWrongBed
          (ReminderStage[LyingWrongbed*2 + 1]  ≠ 0);
#assert SmartBedroom() ⊨
          □ (LyingWrongBed → ◇ RemindedWrongBed);
```

Here, condition *LyingWrongBed* specifies the scenario that someone else is sleeping on patient1's bed, and *RemindedWrongBed* defines the state the reminder is prompted.

Guaranteed Tap_Not_Off Reminder (P2.2): This property states that when the system detects that the shower tap is not off for a long time, the reminder *Tap_Not_Off* will eventually be sent.

```
#define TapNotOff (sensors[ShakeTap] == UNSTATIONARY
          && Duration[ShakeTap]>30);
#define OffTapReminded
           ( ReminderStage[TapNotOff*2] ≠ 0
          || ReminderStage[TapNotOff *2+1] ≠ 0);
#assert SmartShowerRoom() ⊨
          □ (TapNotOff → ◇ OffTapReminded);
```

where condition *TapNotOff* specifies the situation that the shower tap is turned on for more than 30 time units, and *OffTapReminded* defines the state the reminder is prompted.

3) Contradict Knowledge (P3): The following property is specified to check whether there are contradictions in the system. For example, if the PIR sensor is in *SILENT* status, there should be no one in the shower room.

```
#define Contradiction ( Pos_Person[1] == SHOWERROOM
          && sensors[PIR] == SILENT);
#assert SmartShowerRoom() reaches Contradiction;
```

4) Conflicting/False Reminders (P4):

Property	Result	# States	# Transitions	Time(s)
P1.1	-	-	-	OOM
P1.2	True	1.43M	2.04M	815
P1.3	True	10.8M	15.8M	7045
P2.1	True	1.60M	2.43M	1945
P2.2	False	0.07M	0.131M	39
P2.3	False	2.19M	4.53M	12414
P2.4	False	0.832M	1.66M	729
P2.5	False	4314	5150	1.6
P2.6	True	1.58M	2.38M	1913
P3	True	572	745	0.3
P4.1	True	2446	3036	1.11
P4.2	True	0.01M	0.02M	6.1

Table I: Results of Experiment

False Reminders (P4.1): False reminders are generated prompts that should not be sent to patients. In the following, we specify a situation that the *Sit_Bed_Too_Long* reminder is sent to patient1 but in fact he is not in the bedroom.

```
#define FalseReminder (Pos_Person[1] ≠ BEDROOM
          && ReminderStage[SitBedLong] ≠ 0 );
#assert SmartBedRoom() reaches FalseReminder;
```

Conflicting Reminders (P4.2): In the following, *ConflictReminder* defines a state where two reminders (i.e. *WanderingInSR* reminder and *Shower_No_Soap* reminder) are simultaneously prompted to one patient.

```
#define ConflictReminder
          ( ReminderStage[ShowerNoSoap * 2] ≠ 0
          && ReminderStage[WanderingInSR * 2] ≠ 0);
#assert SmartShowerRoom reaches ConflictReminder;
```

Based on the work of Section A and B, experiments are carried out to formally verify the properties against the system model. The experiments test bed is a PC with Intel Xeon CPU at 2.13GHz and 32GB RAM. The results are shown in Table I, where OOM indicates out of memory.

C. Discovery of Unexpected Bugs

Counterexamples are returned as evidences if the system model violates certain properties. They are of great value to system engineers to debug the system. The set of confirmed bugs are reported as follows which are unexpected by the development team.

1) P2.2 - P2.5: The violation of these properties reveals a critical problem of the system that it fails to monitor the patient's location correctly. A patient exiting the shower room with tap left on is a typical case. The two reminders, *Shower_Not_Off* and *WanderingInSR* will repeatedly prompt even though there is no one in the shower room.

2) P3: The verification result shows the contradiction state exists and this exposes the inconsistencies in the system. One possible cause is the failure of location monitoring.

3) P4.1: This property is witnessed to be valid. Through careful investigation, we notice that the rule defined for *Sit_Bed_Too_Long* does not have an identity attached to the rule's condition and hence this reminder is sent to the bed's default owner irregardless of the bed's current user.

4) P4.2: It is validated by the scenario of a patient wandering in the shower room and triggering the *WanderingInSR* reminder. He then ignores the reminder and turns on the shower tap to play with water (A typical behavior of a dementia patient). The water runs for a long time that the *Shower_No_Soap* reminder is triggered, therefore causing the system to prompt the conflicting reminders.

Due to page limits, we skip the detailed feedbacks from the system designers. In general, they improved their system by amending the rules with necessary identify information. Furthermore, in order to precisely detect the patient's location, they added PIR sensors in the bedroom and some rules to assure the consistence among context variables.

D. Discussion

We gained several observations from this case study. First and foremost, model checking techniques can provide a very good guide on system design. From our experiences of working with designers of the system, they usually focus on setting up a demonstration based on selected scenarios without considering other useful situations. In fact, the development and consideration of all possibilities when constructing scenarios and rules is an impossible task and would either take many man-hours to find out through actual deployment. Besides, it is important to find unexpected bugs based on the stakeholders requirements before deployment of the whole system. Hence the engineers can retrieve certain normal or abnormal scenarios they are interested in based on our analysis results.

On the other hand, the experimental results also reflect typical state space explosion problem in model checking techniques. The number of states in verifying property *P1.3* reaches the level of 10^8, which is the limit of explicit-state model checkers like SPIN and PAT. Advanced techniques such as partial order reduction and compositional verification are desirable to alleviate this problem.

VI. Related Work

Pervasive computing systems have achieved many milestones in recent years. However, works on applying formal methods to assure the correctness of such systems are limited. In [4], they proposed a TCOZ model for a smart meeting room system which very well captured the concurrent communications and real-time constraints of sensors and actuators. Important properties are manually proved in the paper. Researchers in [5] used Ambient Calculus to model a location sensitive smart guiding system in a hospital. The mobility issue is well modeled and reasoned in their work. However, both of the two languages cannot model the hierarchies in systems. Moreover, lack of verification tools support restricts the applicability of their approaches to large pervasive computing systems. Our work advances them by adopting hierarchical modeling languages which is also supported by popular model checkers for automatic verification. In [20], the Adaptation Finite-State-Machine (A-FSM) is proposed for modeling context-aware adaptive mobile applications. They also proposed fault patterns based on the A-FSM which can be automatically detected using their algorithms. However, how to model systems in A-FSM is not clear in their work and their approach cannot handle liveness properties. Our work provides modeling patterns for most parts of pervasive computing systems and steps of building a system model. Besides, a wide range of properties can be verified using our approach regarding the safety and liveness requirements.

VII. Conclusion

In this work, we propose a formal modeling framework for pervasive computing systems. Different modeling patterns are discussed according to the typical features of systems such as concurrent communications, context-awareness and layered architectures. We also provide environment modeling patterns which are usually not considered in modeling complex systems. Furthermore, critical properties of safety and liveness requirements are identified and specified in proper logics such as specifying guaranteed reminder services using LTL. To demonstrate our approach, we present a case study of applying the modeling framework to a healthcare system for dementia patients. Critical properties are verified using PAT model checker with unexpected bugs revealed. Experimental results and sources of the bugs are explained. This work demonstrates the usefulness of formal methods (particularly model checking techniques) in analyzing pervasive computing systems. In the future, we will optimize the verification algorithms and explore advanced techniques to tackle the state space explosion problem.

Acknowledgment

The authors would like to thank Lee Vwen Yen Alwyn, Clifton Phua, Zhu Jiaqi and Kelvin Sim from Institute for Infocomm Research in Singapore for the kindness contributions and valuable feedback to this work.

References

[1] D. Saha and A. Mukherjee, "Pervasive computing: A paradigm for the 21st century," *Computer*, vol. 36, pp. 25–31, 2003.

[2] W. K. Edwards and R. E. Grinter, "At home with ubiquitous computing: Seven challenges," in *UbiComp*, 2001, pp. 256–272.

[3] J. Sun, Y. Liu, J. S. Dong, and C. Chen, "Integrating specification and programs for system modeling and verification," in *TASE*, 2009, pp. 127–135.

[4] J. S. Dong, Y. Feng, J. Sun, and J. Sun, "Context Awareness Systems Design and Reasoning," in *ISoLA*, 2006, pp. 335–340.

[5] A. Coronato and G. D. Pietro, "Formal specification of wireless and pervasive healthcare applications," *ACM Trans. Embed. Comput. Syst.*, vol. 10, pp. 12:1–12:18, 2010.

[6] B. Mahony and J. S. Dong, "Blending Object-Z and Timed CSP: an introduction to TCOZ," in *ICSE '99*, 1998, pp. 95–104.

[7] L. Cardelli and A. D. Gordon, "Mobile ambients," in *FoSSaCS '98*, 1998, pp. 140–155.

[8] M. Arapinis, M. Calder, L. Denis, M. Fisher, P. D. Gray, S. Konur, A. Miller, E. Ritter, M. Ryan, S. Schewe, C. Unsworth, and R. Yasmin, "Towards the verification of pervasive systems," *ECEASST*, vol. 22, 2009.

[9] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, *Model checking*. MIT Press, 1999.

[10] J. Biswas, M. Mokhtari, J. S. Dong, and P. Yap, "Mild dementia care at home - integrating activity monitoring, user interface plasticity and scenario verification," in *ICOST*, 2010, pp. 160–170.

[11] J. Sun, Y. Liu, J. S. Dong, and J. Pang, "Pat: Towards flexible verification under fairness," in *CAV '09*, 2009, pp. 709–714.

[12] Y. Liu, X. Zhang, Y. Liu, J. Sun, J. S. Dong, J. Biswas, and M. Mokhtari, "Technical Report for Formal Analysis Pervasive Computing Systems," http://www.comp.nus.edu.sg/~yanliu/techreport.pdf.

[13] P. C. Olveczky and S. Thorvaldsen, "Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in real-time maude," *Theor. Comput. Sci.*, vol. 410, pp. 254–280, 2009.

[14] J. Sun, Y. Liu, J. S. Dong, Y. Liu, L. Shi, and E. Andre, "Modeling and verifying hierarchical real-time systems using stateful timed csp," in *The ACM Transactions on Software Engineering and Methodology (TOSEM)*, 2011, to appear.

[15] R. Alur, "Timed automata," *Theor. Comput. Sci.*, vol. 126, pp. 183–235, 1999.

[16] J. Sun, S. Z. Song, and Y. Liu, "Model checking hierarchical probabilistic systems," in *ICFEM*, 2010, pp. 388–403.

[17] M. Kwiatkowska, G. Norman, and D. Parker, "PRISM 4.0: Verification of probabilistic real-time systems," in *CAV*, 2011, pp. 585–591.

[18] W. Marrero, E. Clarke, and S. Jha, "Model Checking for Security Protocols," Carnegie Mellon University, Tech. Rep., 1997.

[19] K. Du, D. Zhang, X. Zhou, and M. Hariz, "Handling conflicts of context-aware reminding system in sensorised home," *Cluster Computing*, vol. 14, pp. 81–89, March 2011.

[20] M. Sama, S. Elbaum, F. Raimondi, D. S. Rosenblum, and Z. Wang, "Context-aware adaptive applications: Fault patterns and their automated identification," *IEEE Trans. Softw. Eng.*, vol. 36, pp. 644–661, 2010.