1-2014

Towards formal modelling and verification of pervasive computing systems

Yan LIU

Xian ZHANG

Yang LIU

Jin Song DONG

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

See next page for additional authors

Citation

LIU, Yan; ZHANG, Xian; LIU, Yang; DONG, Jin Song; SUN, Jun; BISWAS, Jit; and MOKHTARI, Mounir. Towards formal modelling and verification of pervasive computing systems. (2014). *Lecture Notes in Artificial Intelligence*. 8780, 62-91.
Available at: https://ink.library.smu.edu.sg/sis_research/4981

Author
Yan LIU, Xian ZHANG, Yang LIU, Jin Song DONG, Jun SUN, Jit BISWAS, and Mounir MOKHTARI

Towards Formal Modelling and Verification of Pervasive Computing Systems

Yan Liu[1](✉), Xian Zhang[1], Yang Liu[2], Jin Song Dong[1], Jun Sun[3], Jit Biswas[4], and Mounir Mokhtari[5]

[1] School of Computing, National University of Singapore, Singapore, Singapore
{yanliu,zhangxi5,dongjs}@comp.nus.edu.sg
[2] School of Computer Engineering, Nanyang Technological University, Singapore, Singapore
yangliu@ntu.edu.sg
[3] Singapore University of Technology and Design, Singapore, Singapore
sunjun@sutd.edu.sg
[4] Networking Protocols Department, Institute for Infocomm Research, Singapore, Singapore
biswas@i2r.a-star.edu.sg
[5] CNRS-IPAL, Institut TELECOM, Paris, France
Mounir.Mokhtari@it-sudparis.eu

Abstract. Smart systems equipped with emerging pervasive computing technologies enable people with limitations to live in their homes independently. However, lack of guarantees for correctness prevent such system to be widely used. Analysing the system with regard to correctness requirements is a challenging task due to the complexity of the system and its various unpredictable faults. In this work, we propose to use formal methods to analyse pervasive computing (PvC) systems. Firstly, a formal modelling framework is proposed to cover the main characteristics of such systems (e.g., context-awareness, concurrent communications, layered architectures). Secondly, we identify the safety requirements (e.g., free of deadlocks and conflicts) and specify them as safety and liveness properties. Furthermore, based on the modelling framework, we propose an approach of verifying reasoning rules which are used in the middleware for perceiving the environment and making adaptation decisions. Finally, we demonstrate our ideas using a case study of a smart healthcare system. Experimental results show the usefulness of our approach in exploring system behaviours and revealing system design flaws such as information inconsistency and conflicting reminder services.

1 Introduction

Pervasive computing(PvC) aims to provide people with a more natural way to interact with information and services by embedding computation into the environment as unobtrusively as possible [1,2]. With the rapid increase of ageing population in all industrialised societies which raised serious problems, smart

R. Kowalczyk and N.T. Nguyen (Eds.): TCCI XVI, LNCS 8780, pp. 62–91, 2014.
DOI: 10.1007/978-3-662-44871-7_3

healthcare systems equipped with PvC technology are greatly needed to assist the independent living of elderly people. Such systems make it possible for elderly people to stay in their homes longer and manage everyday tasks without significant burden for their caregivers [3]. PvC systems are context-aware and adaptable to the evolving environment [4]. 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. As shown in Fig. 1, a typical PvC system usually involves sensors to monitor environment changes, a residential getaway to translate the raw sensor data to low-level contexts, an inference engine to aggregate these contexts and perform user activities recognition and a reminder service to render a proper reminder for a proper user. Consequently, the heterogeneity of technology and massive ad hoc interactions among layers make PvC systems highly complicated [5]. Additionally, various environment inputs and unpredictable user behaviours cause the system behaviours beyond control, especially when multiple users are interacting with the system simultaneously.

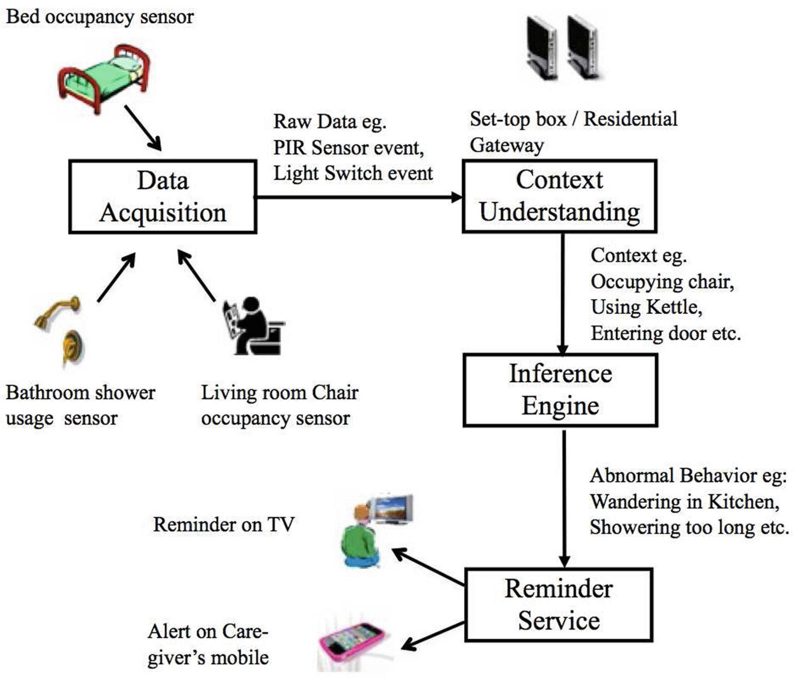

Bed occupancy sensor
Set-top box / Residential Gateway
Raw Data eg. PIR Sensor event, Light Switch event
Data Acquisition
Context Understanding
Context eg. Occupying chair, Using Kettle, Entering door etc.
Bathroom shower usage sensor
Living room Chair occupancy sensor
Inference Engine
Abnormal Behavior eg: Wandering in Kitchen, Showering too long etc.
Reminder on TV
Reminder Service
Alert on Care-giver's mobile

Fig. 1. A typical PvC system

Therefore, it is a challenging task to guarantee the correctness of such systems. Traditional validation methods such as simulation and testing have their limitations in performing this task. By nature, these methods only cover partial system behaviours based on the selected scenarios. Nevertheless, it is costly and time consuming to perform testing on practical systems which may require all sensors to be deployed and normal persons to act like real users. Furthermore, it is impossible to generate all kinds of environment inputs and simulate the user behaviours, especially when more than one users are interacting with the system. Last but not least, if an error is found, it is very hard to pinpoint the source since faults could appear in multiple layers such as the hardware fails to response, a reasoning rule is falsely defined or a bug is in the software system.

In order to overcome these limitations, we propose to use formal methods in the early design stage to analyse PvC systems. By a proper abstraction of the system, we are able to formally describe the system behaviours and user actions using current modelling constructs. Based on the modelling, the correctness and safety requirements can be formally specified as properties that are verifiable against the model. Existing verification techniques are reused to validate the properties by an exhaustive search of the complete system states. Counterexamples are generated to give clues for debugging. The contributions of our work are four-folds as explained below.

Firstly, we propose a framework to formally model the system design and the environment inputs. Important characteristics of PvC systems such as context-awareness, layered architecture and concurrent communications are discussed. Modelling patterns for these features are provided and illustrated with examples. In this work, we adopt CSP# [6] as the exemplar modelling language for its rich set of syntax and many extensions. Dong et al. [7] and Coronato et al. [8] proposed to model such systems using TCOZ [9] and Ambient Calculus [10] respectively. Although these languages are good at modelling the communications and mobility features respectively, the support for modelling 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 propose critical properties and their specification with regards to the correctness requirements from stakeholders (system designers and end users). In the state of art, Arapinis et al. in [11] 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 critical 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 design model by using automatic verification techniques like model checking [12]. Hence, design flaws can be detected at the early design stage.

Thirdly, based on the modelling framework, we propose an approach of automatic rules verification. Rules in PvC systems play a critical role. They are used to aggregate and reason the context data and decide on the adaptation decisions. As shown in Fig. 1, based on the contexts that person in the shower room, using shower tap and tap on for 30 min, the abnormal behaviour, "showering too long" is recognised. The rule defined for this behaviour is triggered and the adaptation decision is to prompt a reminder asking the person to stop showering. Generally speaking, these rules decide the responsive behaviours of the system to the users. Thus, it is essential to assure the correctness of these rules. In the literature, there is very limited work, e.g., [13,14] on rules verification in PvC domain. Most of these existing works are not directly applicable to our system. It is because they are limited to syntactic checking of relations between rules. The contribution of our work lies in redefining rule anomalies based on their execution behaviours and detecting these anomalies with the changing knowledge base.

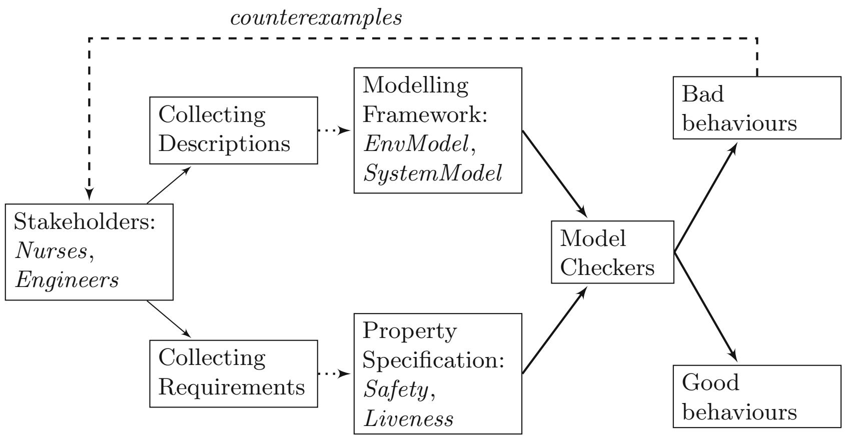

counterexamples
Collecting Descriptions
Modelling Framework: EnvModel, SystemModel
Bad behaviours
Stakeholders: Nurses, Engineers
Model Checkers
Collecting Requirements
Property Specification: Safety, Liveness
Good behaviours

Fig. 2. Formal analysis workflow

Finally, we demonstrate the usefulness of our approach using a case study of a smart healthcare system for mild dementia patients, AMUPADH [15]. A typical workflow of this formal analysis process is shown in Fig. 2. We start the project with collecting requirements through multiple visits to the nursing home and interviews of nurses/doctors. From discussions with system designers, we learn that AMUPADH is a typical PvC 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 behaviours and system design based on our modelling framework using CSP# language. Critical properties such as deadlock freeness, guaranteed reminder service and conflicting reminders tests are verified using PAT model checker [16] Multiple unexpected bugs such as information inconsistency are exposed.

Rest of the paper are organised as Sect. 2 introduces the motivating example AMUPADH system; Sects. 3 and 4 demonstrates our modelling framework of PvC systems and specifications of critical requirements for verification; Sect. 5 presents the rules verification using our modelling framework. Case study on AMUPADH comes in Sect. 6. Section 7 discusses the related work while Sect. 8 concludes the paper with future directions.

2 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 (AAL) systems equip the environment with a spectrum of computation and communication devices that seamlessly augment human thoughts and activities. AMUPADH is an AAL system deployed in a Singapore Based nursing home, Peaceheaven Nursing Home[1]. It is able to monitor the patients' behaviours using activity recognition techniques (sensors and reasoning rules) and offer help to the patients (prompt reminders through actuators such as speakers etc.).

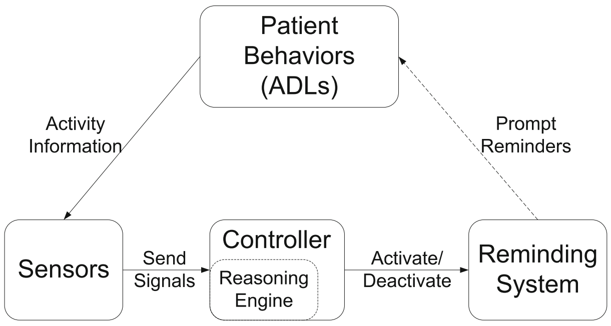


Fig. 3. An overview of the smart bedroom system

2.1 System Overview

The architecture of the system is shown in Fig. 3. The system is deployed in a room with two beds and a shower room. Different kinds of sensors are deployed 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 middleware via wireless network. The *controller* in the middleware translates 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[2] which is a rule language 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 recognised, the system will prompt a reminder requesting him to use his own bed.

[1] Located at 9 Upper Changi Road North, Singapore, 507706. Tel: +65-65465678.
[2] Drools Expert: http://www.jboss.org/drools/drools-expert.html

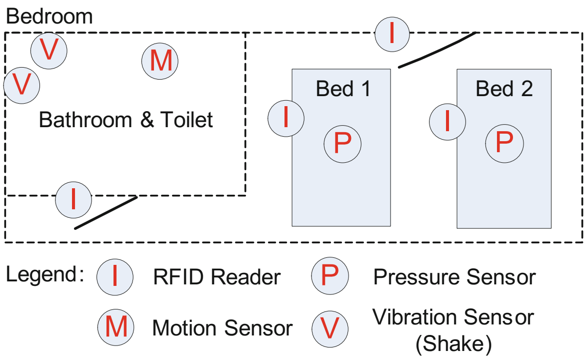


Fig. 4. Sensor layout in the bedroom

2.2 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 Fig. 4.

- **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.

2.3 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 min. The value is carefully tuned by system engineers in consideration of energy saving and slow user movements in AMUPADH system. The context checker is an additional step to make sure the consistency of contexts. 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 min. The messages are sent out via a shared bus. The full set of 23 rules used in the system is listed in [17].

2.4 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 reach the maximum number, an alert will be sent to nurses.

3 A Modelling Framework for PvC Systems

The general picture of a PvC system is shown in Fig. 5. The system adopts a layered design and seamless interacts with the environment. Modelling of a PvC systems involves not only the modelling of important features of each important component but also the modelling of the environment inputs which play an important role in PvC systems but is often ignored in most system models.

3.1 Modelling Environments

PvC systems seamlessly interact with the environments and acquire context inputs from the users and objects like TVs and Beds. PvC systems are often driven by the environment context change (we call it *scenario* here). For example, a person entering an empty room 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 analysing verification results.

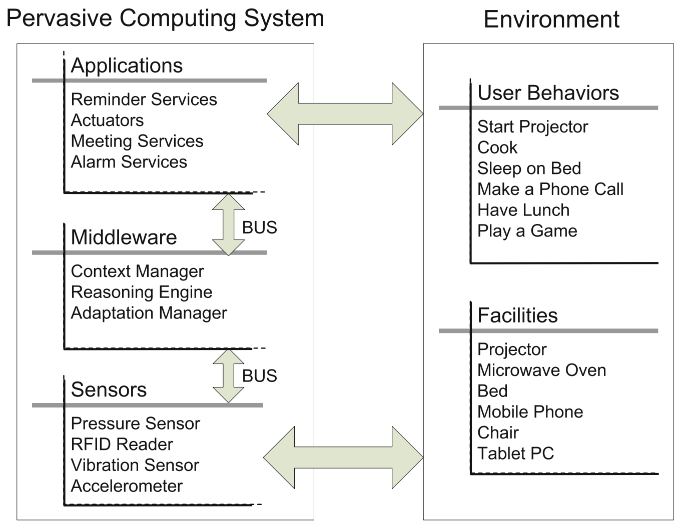


Fig. 5. Architectures of PvC systems

Modelling Activities and Environment Objects. User behaviours are various and usually unpredictable. For most PvC 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, after entering a room, a person 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 modelling patterns.

In the shower room, a user performs many activities such as wandering or turning on the shower tap. These activities can be modelled as *events* which are abstractions of the observations. For example, an activity represented as event *exitShowerRoom* is an observation of the user's behaviour 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 modelled 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 chooses 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 behaviour is synchronised with the status of the object being used. In fact, it is essentially the problem of modelling synchronous behaviours. We propose to use event synchronisation in CSP# and give an example of shower tap model in the following. Other ways of modelling such as using a global variable or synchronous channels are also possible.

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

The constraint of using tap behaviours is modelled 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 synchronised on common events appearing in all of them. Interested readers can refer to [6] for more details. Here, the *turnOnTap* event becomes a common event between the two processes.

Modelling Location Transitions. While modelling the patients behaviours, 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 modelling approach is to reflect the location transitions in the model and to generate realistic sequences of activities.

Modelling 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 behaviour 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 behaviour 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.

3.2 Modelling System Design

PvC systems share the features such as layered architecture and concurrent communications. In the following, we discuss these common features and their modelling layer by layer.

Modelling 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 s while pressure sensor sends every 10 s. 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 PvC 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 [18]. The details about signal encoding/decoding and message transmission via wireless networks are abstracted away for simplicity in our work.

Modelling Concurrent Interactions. Sensors interact with the environment by detecting events and report sensed contexts by transmitting signals to middleware. The behaviours of detecting and transmitting can be abstracted to two modelling patterns which are synchronous events and message passings respectively. Event synchronisation has been introduced in Sect. 3.1. As for message passing, there are different modelling patterns in different languages. Some languages support synchronous channels through which the sending and receiving events are synchronised. 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 synchronisation that the event *turnOnTap* can be synchronised in all three processes.

Modelling 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 analyse these behaviours, we propose to use timed modelling languages such as Stateful Timed CSP (STCSP) [19]. Timed Automata (TA) [20] is not suitable in this case because the hierarchal modeling is not supported in TA. The modelling 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 sensing behaviours of the pressure sensor which senses the pressure on the bed for every 10 time units. Its status is transmitted immediately after the sensing.

Modelling Sensor Failures. Sensors have limited accuracy that they may fail to detect certain events. They could also run out of battery and fail to send the signals. Intuitively, we model this with probabilistic modelling constructs, e.g., Probabilistic CSP# (PCSP#) [21], Probabilistic Timed Automata (PTA) [22].

```
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 modelling 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 modelled using different language constructs. Notice that the two modelling 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.

Modelling Middleware Layer. As shown in Fig. 5, 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 modelled using shared variables in supporting modelling 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 Sect. 2.3 in CSP#:

```
Rule() = if(sensors[Pressure_Sensor] == SITTING &&
     Duration[Pressure_Sensor] > 30){
     res!Act.SitTooLong.PersonA → 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 modelled 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*.

Modelling 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 modelling of how the adaptation decisions are executed. For instance, in the AMUPADH system, the reminding system is modelled as follows:

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

By decoding the message received from the 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.

3.3 Compose a Complete Model

In PvC systems, different components in different layers cooperate to fulfil the system goals. However, how to model this cooperate relations are left to be discussed till now. From a careful study, we discover that in PvC systems, there are three common types of relationships between system 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 synchronised behaviours. These relations can be well supported in hierarchical languages such as CSP#. Respectively, these three relations can be modelled 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 modelling 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.

Choosing a Modelling Language. The above mentioned modelling patterns are supported in most modelling languages of CSP family. It is also possible to be translated to other formalisms e.g., Timed Automata. When it comes to unify concurrent modelling with probabilistic or real time modelling or both, there exists some approaches. For example, in PAT framework, CSP# supports for modelling concurrent system behaviours; PCSP# extends CSP# with probabilistic behaviour modelling, it is suitable to model failures in PvC systems; RTCSP# extends CSP# with real time constructs which can be used to model the periodic sensing behaviour; and finally PRTS which integrates both probabilistic and real time modelling constructs under one roof. However, it is important to choose a proper modelling language according to different targets. For example, CSP# is most suitable for reasoning concurrent behaviours of PvC systems while PRTS will be an over cure. We may also argue that it requires minimal effort to extend a CSP# model to an PCSP# model and likewise.

4 Scenario Verification

After system engineers finished the design of a PvC system, they are often asked to provide guarantees for correctness and 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, using 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.

4.1 Desirable Properties

Properties regarding the good behaviours of the systems are desirable.

Deadlock Freeness. Deadlock freeness is one of the important safety requirements and should be assured before checking any liveness properties. 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 checkers.

Guaranteed Services. Well designed application services determine fundamental responsive behaviours of pervasive healthcare systems. For example, in a smart meeting room, upon detection of some one entered the room, a 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,

$\Box(\texttt{PatientWandering} \rightarrow \Diamond\ \texttt{LeaveRoomReminder})$

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 modelling languages such as STCSP.

Security. Since PvC 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,

$\Box(\texttt{FoodDeliveryPerson} \rightarrow \texttt{not}\ (\Diamond\ \texttt{AccessPatientProfile}))$

Model checking techniques for security problems are proposed in papers such as [23].

4.2 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.

System Inconsistency. Failures of sensors and wireless networks may cause contexts of the environment in the 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.

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 [24]. 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.

5 Rules Verification

PvC systems are widely applied in healthcare domain, especially in the area of assistive living. In fact, it is challenging to automatically recognise activities of assisted people and to render adequate assistive services. In the current literature, rule based system design is adopted that it is able to provide dependable assistance services based on sensors integrated into the living ambient environment [25,26]. In such systems, rules are manually defined by system engineers based on observations from doctors and caregivers. As introduced in Sect. 2.3, a rule consists of a name, a condition field and an action field. In the condition, a certain activity to be monitored is defined based on contexts such as status of sensors, duration of sensor readings or system flag variables. The assistive service is an adaptation decision which is defined in the action field. During the reasoning process, all the rules will be evaluated based on the current contexts and actions will be executed upon the satisfaction of the rule's condition. Working in such a fashion, rule based system is able to intelligently recognise activities of users and adapt to their needs accordingly.

However, the correctness of the rules remains a non-trivial problem. Anomalies such as duplication, unreachable condition and conflicts widely exist in rule bases. Due to rule engineer may have limited knowledge of assisted user, incorrect or vague rules may be defined which will impair the system's capability in determining activities. The accuracy of activity information is lowered that it may further result in a lack of service to be offered. What's more, unreliable rules would also provide a misleading reflection of the actual situation, which is unacceptable in mission-critical or urgent scenarios. Besides, to verify relatively large rule repositories is considerably laborious. Therefore there is a need to construct an approach that is able to verify and ensure the specificity of rules, and to also provide evidence of the erroneous rules.

In this work, we propose the definition and specification of rule anomalies according to their behaviours and influence on the system behaviour (instead of using common definitions based on the syntax and semantics). By reusing the system model constructed in Sect. 3, we are able to detect rules anomalies feasibly using existing model checking algorithms. In the following, we list the three types of rule anomalies.

5.1 Non-reachable Rules

Non-reachable rules are trivial as some rules' conditions are never satisfied during all system runs. These rules can be unintentionally introduced by rule developers. Although the system's correctness is unaffected, they add complexity to the model and slow down the rules evaluation process. On the other hand, it could be the reason that a critical context used in the rule condition is always not available. This, in fact, means the system fails to detect certain events in the environment. For example, in the scenario of detecting the usage of cupboard, the rule defined for this behaviour is never fired because the sensor engineers forget to deploy a reed switch sensor (which is used for detecting open/close action). In such a case, detecting these rules will reveal critical problems of the system.

Detection of non-reachable rules can be done by reachability checking. Based on the system model, we analyse all the system states for each rule individually to see whether its condition can be satisfied or not. The pattern proposed for expressing this property is as follows.

```
assert rule_SBTLAA.condition reachable
```

We take the rule in Sect. 2.3 as an example. The rule name is represented as rule_SBTLAA (where SBTL_AA stands for sitting on bed too long for person A on Bed A). By defining its condition as a state, we try to assert whether this specific state is reachable or not. During the verification process, each system state will be compared to see if there is a match by using existing reachability checking algorithms. Non-reachable rules are better to be eliminated before checking other rule anomalies.

5.2 Redundant Rules

Redundant rules are occurrences of multiple rules firing together at same system states and producing non-conflicting system results. In fact, the rule system is usually maintained by multiple engineers, even end-users. It is often the case that they put similar rules into the system such as similar condition with different parameters or different actions. Redundant rules will increase the complexity of the rules and slow down the rule execution process. Furthermore, redundant rules create redundant information that blows up memory easily.

We define two kinds of redundant rules, *duplicated rules* and *subsuming rules*. The former refer to rules that always fire together at the same time and have a non-conflicting actions. Identical rules where their conditions and actions are both the same is considered as one special case of duplication. The latter applies to the case that one rule is always fired with the other rule where their actions are not contradict. Thus, the scenario covered by the first rule is included in the second one.

Redundant rules can be specified as LTL formulae. By the above definition, we propose the specification patterns in LTL as follows:

```
□(rule1.condition → rule2.condition)                        (1)
□(rule2.condition → rule1.condition)                        (2)
```

In the example, if (1) and (2) both turns out to be true, then we say rule1 and rule2 are duplicates. If only one of them is true, for instance, (1) is true, then rule1 is subsumed by rule2 where the scenarios at which rule1 satisfies is covered by rule2 as well.

5.3 Conflicting Rules

Conflict anomalies focus on the actions of rules. In a particular state, two rules both fires but with contradicting actions triggered. Then, they are considered as conflicting rules. This type of anomalies is usually not because of careless human errors, but because of limitations in the rule design. In fact, engineers define rules based on their limited knowledge of the actual user behaviour. However, it is impossible for them to figure out all the scenario. Especially in the case of dementia patient caring, the abnormal behaviour is beyond the imagination of normal people. Thus, contradictions often happen in the system. Furthermore, conflict rules are critical but difficult to detect. They could cause the user to be confused which may be harmful to their health or even life. But conflicting situations are not easily revealed during lab testing where only selected scenarios are tested. Thus, using advanced techniques such as model checking which can simulate every possible behaviour of user and perform complete search of state space are favourable.

In an attempt to reuse existing techniques, we detect conflicting rules in two ways: (1) we perform verification of Sect. 5.2. Based on the verification result, we inspect the actions of the redundant rules to find conflicting cases; (2) by defining impermissible sets which contains contradict knowledge, we check if such occasion can be reached during the system run. Note that, since the contexts only kept in the knowledge base of rule systems, contradictory contexts must imply contradicts in the rules. By performing reachability checking, we are able to find conflicts with witness traces revealing the rules which lead to the conflicting state. We take an example in AMUPADH system.

```
impermissible_set (Loc_PersonA = ShowerRoom, Status_PersonA = Sleeping)
assert impermissible_set reachable
```

In the example, we define an impermissible set says person in the shower room and person sleeping cannot be true at the same time. These two contexts both are high level contexts that are generated by rules. Thus, if two elements in the impermissible set becomes true at the same time in the model, the two rules which generate them must be conflicting in a particular scenario.

In [27], a rule modelling approach based on language translation is constructed to automate the process of rules verification. The correction strategies for rule anomalies are also proposed.

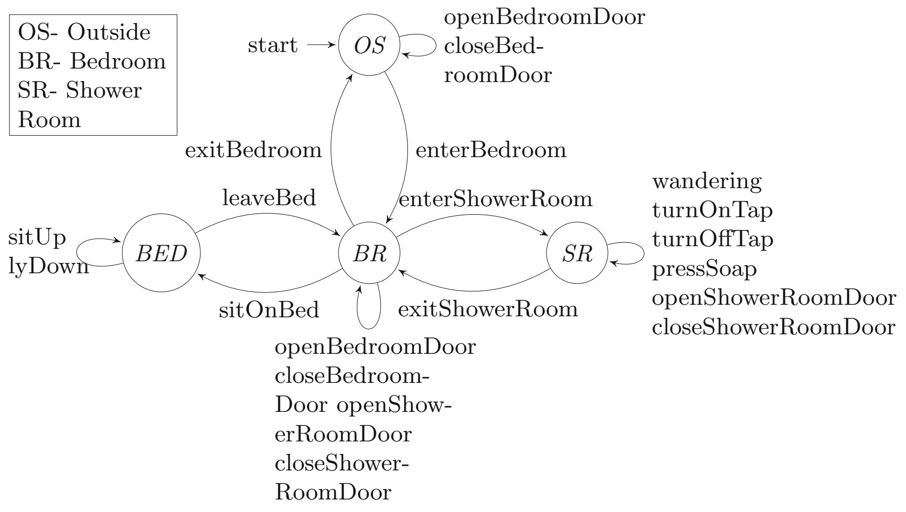


Fig. 6. Patient behaviours

6 Case Study: Formal Analysis of AMUPADH

The proposed approach is applied to analyse AMUPADH. We adopt CSP# modelling language since it supports most of the modelling 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.

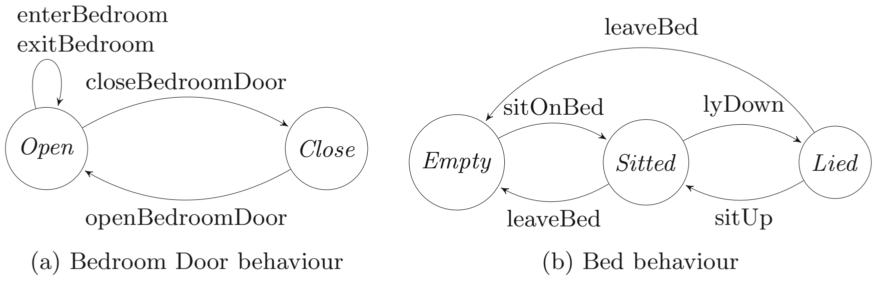


Fig. 7. Surrounding environment

6.1 System Modelling

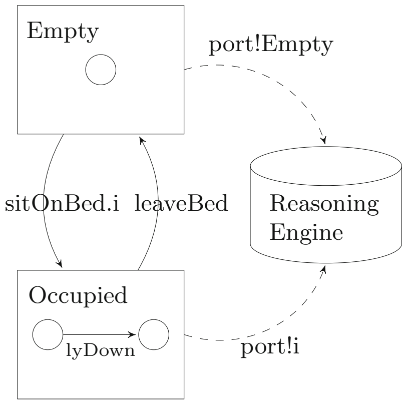
In this section, we model the environments and the system design using our framework and use Labeled Transition Systems (LTS) for demonstration.

Environment Model. As shown in Figs. 6 and 7. These LTSs can be generated using simulation function of PAT. In Fig. 6, there are four possible locations that a patient can reside. The transition edges between states are labeled with patient's activities.

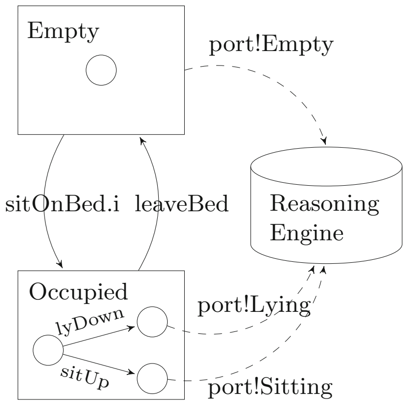
This patient model should be synchronised with objects within the surrounding environment. The objects that are modelled include doors of bedroom and washroom, beds and washroom taps. The behaviour models of the doors and beds are shown in Figs. 7a and b respectively.

Sensor Model. Different sensors are used in AMUPADH to monitor specific behaviours 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 synchronised channel *port*. Every sensor possesses multiple unique states when made available to the system. Figure 8 shows the modelling of sensors using the bed RFID readers and bed pressure sensors as mentioned in Sect. 2.2. 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();
```



(a) Bed RFID Reader



(b) Bed Pressure Sensor

Fig. 8. Sensor behaviours

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 process *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*. Then, the *FireAllRules* process is invoked to perform rules evaluation. The syntax $call(setTimer, id, status, Duration)$ in the above model is used to call an external static function *setTimer* (written in C#). *Duration* be will updated externally according to the input of sensor *id* and *status*. This is a special feature in PAT, which allows users to separate 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 modelling. The process $Rule\_14\_1()$ models a complicated rule defined for recognising the wandering behaviour 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()};
...
```

Reminding System Model. In the system, reminders are activated/deactivated upon receiving corresponding messages from the controller. As shown in Fig. 9, the reminding system receives a triplet from the controller via channel *res*. This triplet consists of a command, behaviour code and patient ID. If the command is

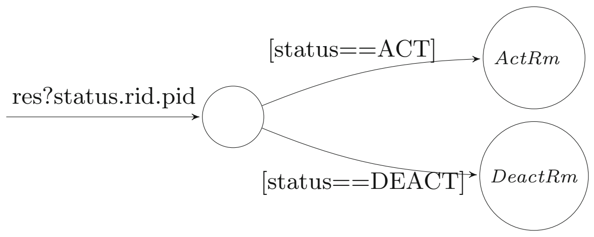


Fig. 9. Reminding system behaviours

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 [17].

6.2 Scenario Verification Experiments

In this section we verify the proposed properties against the system model built in Sect. 6.1. He experiments test bed is a PC with Intel Xeon CPU at 2.13 GHz and 32 GB RAM.

Deadlock Freeness. Since each layer of the system as well as the environment model are independent from each other except for channel communications, we conducted the experiments incrementally. During verification of a particular component model, we abstract away the details of other component models leaving only the channels for receiving messages. Doing in this way, we are able to check deadlock freeness locally for all system components and keep the composition of component models in a manageable level. In the Table 1[3], the row starts with *env* represents the environment model; row $env + snr$ represents the model composed by environment model and sensor model; row $env + snr + mdw$ adds middleware model into previous one; and the last row is the complete model with all components. It turns out to be that the complete model including bedroom and shower room scenario is too large for verification. We split it into two sub-models according to the locations. The experiment results show the rapid increase of state space when more components are composed.

In CSP#, a deadlock freeness property is specified as

```
#assert CompleteSmartRoom() deadlockfree;
```

[3] St- States, OOM- Out of Memory.

Table 1. Results of deadlock freeness checking

Model	Bedroom		ShowerRoom		BothRooms	
	#St/k	Time/s	#St/k	Time/s	#St/k	Time/s
env	0.028	0.005	0.008	0.005	0.082	0.010
env + snr	0.157	0.080	0.072	0.030	0.906	0.339
env + snr + mdw	17.40	7.799	56.01	23.00	8319	4017
Complete	731.5	384.2	7059	4031	OOM	OOM

Guaranteed Reminders. Guaranteed reminders are important measurements of the system, which are illustrated using the patterns of service effectiveness. We take the reminder services of *Lying_Wrong_Bed* in bedroom as examples. Other properties for guaranteed reminder services can be specified similarly.

```
#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. This property states that when a patient is sleeping in a wrong bed, the system will always prompt the *LyingWrongBed* reminder eventually. The results of the verification are shown in Table 2. The first two reminders are checked against the bedroom system model while the rest are against shower room model. Surprisingly, all the reminders on shower room fails and it takes variant time to invalid a property due to the depth of the bugs.

Table 2. Results of guaranteed reminders checking

Property	Result	# States/k	Time/s
LyingWrongBed (LWB)	True	808.4	616.8
SitBedTooLong (SBTL)	True	798.3	607.2
ShowerNoSoap (SNS)	False	196.6	107.5
ShowerTooLong (STL)	False	1018	2635
ShowerNotOff (SNO)	False	701.8	489.1
WanderingInSR (WIS)	False	58.24	27.48

Testing of Faults. Various fault occur in AMUPADH system, the most common ones are the inconsistencies, the false reminder and reminder conflicts. They are introduced in the following. Experiment results shown in Table 3 reveals multiple bugs.

Table 3. Results of testing faults

Model	Fault type	Result	# States/k	Time/s
Bedroom	FalseAlarm: LWB	False	731.5	371.7
	FalseAlarm: SBTL	True	1.463	0.479
	CR: LWB vs. SBTL	True	20.6	7.89
Shower Room	InConsistency	True	0.404	0.180
	CR: SNS vs. WIS	True	10.34	4.150
	CR: SNS vs. STL	True	20.98	7.898
	CR: SNS vs. WNO	True	10.54	3.660
	CR: STL vs. SNO	True	16.35	5.785
	CR: STL vs. WIS	True	16.35	5.767
	CR: WIS vs. WNO	True	5.2	1.758

Inconsistent Knowledge. In the shower room, it is the case that there is no one in the room (the PIR sensor indicates *SILENT* status), while the system variable recording patient 1's location remains to be in the room. This property is specified as follows:

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

False Reminders. 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 (CR). 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;
```

6.3 Detecting Rule Anomalies in AMUPADH: Experiments

We perform rules verification in the following steps.

Parsing Drools Rule to CSP#. Manually modelling of all the rules are time consuming and error prone due a large number of rules are defined. Thus, we developed tool for automatically translate Drools rules used in AMUPADH to CSP# syntax in two steps.

Step 1: Extract Shared Information. For the purpose of easy management, the shared information is declared and kept in separate files from rule files. We need to first extract these information. Fortunately, customised data type and external function calls are supported in PAT. Thus, it is only needed to extend the original Java classes with additional methods for value retuning conform to PAT models. However, rewrite the Java methods to C# codes is a better solution since PAT is written in C#.

Step 2: Mapping Rules into CSP#. The parser processes the rules one at a time and splits the rule into three parts, i.e., rule name, conditions and consequences by reading the keywords *rule*, *when* and *then* respectively. We then map the Drools rule into CSP# by mapping rule names into rule associated comments, the conditions to *ifa* expressions, the consequences into *ifa* statements and point to the next rule in the *else* part (evaluation of the rules is sequential).

The two-step parsing tool automates the process of modelling rules and reduces the time required for verifying rules. Although human intervention might be required in rewriting Java classes into C#, the effort is minimal since Java classes are seldomly changed during development. However, this parser is unable to treat rules with priorities and rule chaining at the moment, extension is possible once needed.

Detetcting Rule Anomalies

Step 1: Define Rule Conditions as States. In order to specify the properties associated with evaluating conditions of rules, we explicitly define all the rule conditions as specific states. We take *Rule*_14_1 in Sect. 6.1 as an example. *Rule*_14_1 is the 14th rule in the rule base defined for person 1.

```
#define Rule_14_1 (sensors[ShakeTap] == STATIONARY &&
      sensors[PirShowerroom] == FIRING &&
      Duration[PirShowerroom] ≥ 15 &&
      !ShowerFlag && Location_Person[1] == SHOWERROOM)
```

Step 2: Specify Rule Anomalies. We check three types of rule anomalies.

– **Non-reachable rules** are rules that cannot be satisfied during all executions of the system. The following property asserts if *Rule*_14_1 is reachable.

```
#asset SmartShowerRoom reaches Rule_14_1;
```

- **Redundant Rules** are rules that have similar effects on the system such as always fires together. We first check the subsumed rules which is the case that one rule always fires with the other rule. An simple example property specification is as follows.

  ```
  #asset SmartShowerRoom ⊨ □ (Rule_1 → Rule_2);
  #asset SmartShowerRoom ⊨ □ (Rule_2 → Rule_1);
  ```

  By checking every pair of rules, we are able to do a thorough testing to find all possible subsumed rules. Duplicated rules are two rules subsumed with each other.
- **Conflicting Rules** are rules that fires together at a particular state but have conflict consequences. To detect this anomaly is to define an impermissible set like the following example.

  ```
  #define contradict_state (ShowerFlag == true &&
          Location_Person[1] != SHOWERROOM &&
          Location_Person[2] != SHOWERROOM);
  #assert SmartShowerRoom reaches contradict_state;
  ```

  This *contradict_state* says someone is taking shower, but neither of the two patients is in the shower room. Note that there are only two users in the model and *ShowerFlag*, *Location_Person*[1] and *Location_Person*[2] are all variables updated by some rules. If this contradict state is reachable, there must be some rules conflicting. By inspecting the witness trace reported by PAT, we are able to identify them.

Step 3: Verify Rules Using PAT. Finally, we integrate all the rules into the formal model and verify the properties specified in previous step using built-in verification algorithms in PAT. The results are shown in Table 4 with multiple conflicts found.

Table 4. Results of detecting rule anomalies

Scenario	# Rules	# Non-reachable	# Subsumed/# Duplicated	# Conflict
Bedroom	17	2	8/3	2
Shower Room	22	5	16/2	4
Avg. Time(s)	-	2.05	3.05	-

6.4 Bug Report

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.

System implementation fails to meet requirements

– *Guaranteed Reminders.* This experiment reveals a critical problem of the system that the system 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.

Unexpected Faults Arising out of system complexity

– *False Alarm in Bedroom.* The result of the second 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 regardless of the bed's current user.
– *Conflict Reminders.* From the experiment results, we found many scenarios where there are reminder conflicts. For example, a patient wandering in the shower room tirggers the *WanderingInSR* reminder. He then ignores the reminder and turns on the shower tap to play with water (A typical behaviour 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.

Anomalies in Activity Recognition Rules

– *Non-reachable rule.* In the bedroom scenario, the rule defined to recognise an activity of opening a cupboard is not reachable. The reason is that sensor engineers removed the reed switch sensor on the cupboard without notifying the rule engineers.
– *Redundant rules.* Five duplicated rules are discovered which were accidentally added into the rule repository for testing and were not removed due to negligence.
– *Conflicting rules.* A scenario where a pair of conflicting rules are witnessed that the monitored user have been showering for a long period of time, yet continues to ignore the reminder that prompts him to use the shower foam. This was the reason that led to the triggering of two contradictory reminders that request a user to perform activities in two different locations at the same time, which is physically impossible.

Discussion

Usefulness. We gained several observations from this case study. First, 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. It is not only because of the high cost of hardware devices but also to complete a full demonstration is time consuming. 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. In fact, some of the bugs (e.g., False Alarm) we reported are occurring in execution of AMUPADH system and some of them are unexpected (e.g., inconsistency). The counterexamples reported from the experiment also helped the engineers to pinpoint the source of the bug. 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.

Additionally, we observed the failure of updating the correct location information of the patient leads to the violation of important properties. From the discussion with hardware engineers, we learned that RFID readers have limited detection range. We may think it is unwise to solely rely on RFID readers to track the patients. During the experiments, we also noticed that a lot of redundant messages are sent out by the reasoning engine which increase the complexity of the system and slow down the verification.

Thoughts of solving state space explosion in PvC system verification. The experimental results reflect the typical state space explosion problem. The number of states in checking deadlock freeness of the complete model reaches the level of 10^8, which is the limit of explicit-state model checkers like SPIN and PAT. The state of art state space reduction methods like partial order reduction may not have significant improvement of this problem. Compositional verification on the other hand draws our attention. From the deadlock freeness checking, we noticed that if all components are locally deadlock-free, it is of great possibility that the complete system model which is a composition of all the components is free of deadlock. Obviously verifying a local property of a component is much easier than verify it against the system model. Furthermore, the general architecture of PvC systems suggests that there are almost no sharing recourses between components. The independency between system components further proves that compositional verification could be a feasible solution to state space explosion problem. Thus, in future, we shall explore how composition verification techniques can be applied.

7 Related Work

PvC systems have achieved many milestones in recent years. However, works on applying formal methods to assure the correctness of such systems are limited. In [7], they proposed a TCOZ model for a smart meeting room system which very well captured the synchronised communications and real-time constraints of sensors and actuators. Researchers in [8] used Ambient Calculus to model a location sensitive smart guiding system in a hospital. The mobility issue is well modelled in their work. Important properties are manually proved in both of the two papers. However, both of the two languages does not have support for hierarchical structures. Moreover, lack of verification tools support restricts the applicability of their approaches to large pervasive systems. Our work advances them by adopting hierarchical modelling patterns. Automatic verification of our modelling framework can be supported by popular model checkers.

In [28], Adaptation Finite-State-Machine (A-FSM) is proposed for modelling adaptations between system states in context aware adaptive applications. Fault patterns based on the A-FSM and their detection algorithms are presented as well. However, how to model systems in A-FSM is not clear and liveness properties are not supported in their work. Researchers in [11] proposed multiple important properties regarding security, safety requirements in PvC systems. Formalisation patterns are illustrated and possible verification approaches are explored. However, since they lack the underlying modelling patterns, the properties and their verifications are very difficult to apply. In our work, we further classified the important requirements into safety and liveness properties and formalise them in popular logics which are checkable based on our modelling framework. Besides, we propose scenario verification which verifies critical requirements on an exhaustive enumeration of targeted scenarios which is more focused than aimless, random verification approaches and more complete than verification/testing upon selected cases.

In rules verification, Ligeza and Nalepa [13] proposed definitions for rule anomalies regarding redundancy, consistency, completeness and determinism. Preece et al. [14] surveyed the verification of rule based systems focusing on detecting anomalies. Five rule verification tools are compared based on their capability of detecting rule anomalies such as redundancy, ambivalence etc. However, their definitions for anomalies and the surveyed algorithms are not directly applicable to PvC systems. Most of their algorithms detect anomalies based on syntax checking and semantic logics inspection between rules, instead of how the rules affect the system behaviour. Furthermore, the algorithms are mostly designed for goal-driven (stateless) rules where knowledge is not shared between different rounds of evaluation. This is certainly not the case of how rules working in PvC systems. Thus, in our work, we redefined the rule anomalies according to their influences upon the system behaviours and formulate them into properties which are verifiable on our modelling framework by reusing existing model checking algorithms.

8 Conclusion

In this work, we propose a formal modelling framework for pervasive computing systems. Different modelling patterns are discussed according to the typical features of systems such as concurrent interactions, context-awareness and layered architectures. We also provide environment modelling patterns which are usually not considered in modelling complex systems. Based on the modelling framework, we propose scenario verification where critical properties of safety and liveness requirements are identified and specified in proper logics such as specifying guaranteed reminder services using LTL, and rules verification where rule anomalies are redefined upon system behaviours and formulated to formal properties which can be verified using existing model checking algorithms.

To demonstrate our approaches, we present a case study of an living assisting system for elder dementia patients. We model the system using our modelling framework and conduct experiments of scenario verification and rules verification. Multiple bugs are revealed. Experimental results and sources of the bugs are explained.

This work demonstrates the usefulness of formal methods (particularly model checking techniques) in analysing PvC systems. In the future, we will apply probabilistic model checking techniques for quantitative analysis of PvC systems and explore compositional verification techniques to alleviate 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. We also want thanks the anonymous reviews for their valuable suggestions in improving the manuscript.

References

1. Weiser, M.: The computer for the 21st century. Sci. Am. **265**(3), 66–75 (1991)
2. Estrin, D., Culler, D., Pister, K., Sukhatme, G.: Connecting the physical world with pervasive networks. IEEE Pervasive Comput. **1**(1), 59–69 (2002)
3. Nehmer, J., Becker, M., Karshmer, A., Lamm, R.: Living assistance systems: an ambient intelligence approach. In: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, pp. 43–50 (2006)
4. Saha, D., Mukherjee, A.: Pervasive computing: a paradigm for the 21st century. Computer **36**, 25–31 (2003)
5. Edwards, W.K., Grinter, R.E.: At home with ubiquitous computing: seven challenges. In: Abowd, G.D., Brumitt, B., Shafer, S. (eds.) Ubicomp 2001. LNCS, vol. 2201, pp. 256–272. Springer, Heidelberg (2001)
6. Sun, J., Liu, Y., Dong, J.S., Chen, C.: Integrating specification and programs for system modeling and verification. In: TASE, pp. 127–135 (2009)
7. Dong, J.S., Feng, Y., Sun, J., Sun, J.: Context awareness systems design and reasoning. In: ISoLA, pp. 335–340 (2006)
8. Coronato, A., Pietro, G.D.: Formal specification of wireless and pervasive healthcare applications. ACM Trans. Embed. Comput. Syst. **10**, 12:1–12:18 (2010)
9. Mahony, B., Dong, J.S.: Blending object-Z and timed CSP: an introduction to TCOZ. In: ICSE '99, pp. 95–104 (1998)
10. Cardelli, L., Gordon, A.D.: Mobile ambients. In: Nivat, M. (ed.) FOSSACS 1998. LNCS, vol. 1378, pp. 140–155. Springer, Heidelberg (1998)
11. Arapinis, M., Calder, M., Denis, L., Fisher, M., Gray, P.D., Konur, S., Miller, A., Ritter, E., Ryan, M., Schewe, S., Unsworth, C., Yasmin, R.: Towards the verification of pervasive systems. ECEASST **22**, 1–15 (2009)
12. Clarke Jr, E.M., Grumberg, O., Peled, D.A.: Model Checking. MIT Press, Cambridge (1999)
13. Ligeza, A., Nalepa, G.J.: Rules verification and validation. In: Giurca, A., Gasevic, K.T.D. (eds.) Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches, pp. 273–301. IGI Global, Hershey (2009)

14. Preece, A.D., Shinghal, R., Batarekh, A.: Principles and practice in verifying rule-based systems. Knowl. Eng. Rev. **7**(02), 115–141 (1992)
15. Biswas, J., Mokhtari, M., Dong, J.S., Yap, P.: Mild dementia care at home – integrating activity monitoring, user interface plasticity and scenario verification. In: Lee, Y., Bien, Z.Z., Mokhtari, M., Kim, J.T., Park, M., Kim, J., Lee, H., Khalil, I. (eds.) ICOST 2010. LNCS, vol. 6159, pp. 160–170. Springer, Heidelberg (2010)
16. Sun, J., Liu, Y., Dong, J.S., Pang, J.: PAT: towards flexible verification under fairness. In: Bouajjani, A., Maler, O. (eds.) CAV 2009. LNCS, vol. 5643, pp. 709–714. Springer, Heidelberg (2009)
17. Liu, Y., Zhang, X., Liu, Y., Sun, J., Dong, J.S., Biswas, J., Mokhtari, M.: Technical report for formal analysis pervasive computing systems. http://www.comp.nus.edu.sg/~yanliu/techreport.pdf
18. Olveczky, P.C., Thorvaldsen, S.: Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in real-time maude. Theor. Comput. Sci. **410**, 254–280 (2009)
19. Sun, J., Liu, Y., Dong, J.S., Liu, Y., Shi, L., Andre, E.: Modeling and verifying hierarchical real-time systems using stateful timed CSP. ACM Trans. Software Eng. Methodol. (TOSEM) **22**(1), 3:1–3:29 (2013)
20. Alur, R.: Timed automata. Theor. Comput. Sci. **126**, 183–235 (1999)
21. Sun, J., Song, S., Liu, Y.: Model checking hierarchical probabilistic systems. In: Dong, J.S., Zhu, H. (eds.) ICFEM 2010. LNCS, vol. 6447, pp. 388–403. Springer, Heidelberg (2010)
22. Kwiatkowska, M., Norman, G., Parker, D.: PRISM 4.0: verification of probabilistic real-time systems. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 585–591. Springer, Heidelberg (2011)
23. Marrero, W., Clarke, E., Jha, S.: Model checking for security protocols. Technical report. Carnegie Mellon University (1997)
24. Du, K., Zhang, D., Zhou, X., Hariz, M.: Handling conflicts of context-aware reminding system in sensorised home. Cluster Comput. **14**, 81–89 (2011)
25. Antoniou, G.: Rule-based activity recognition in ambient intelligence. In: Bassiliades, N., Governatori, G., Paschke, A. (eds.) RuleML 2011 - Europe. LNCS, vol. 6826, pp. 1–1. Springer, Heidelberg (2011)
26. Storf, H., Becker, M., Riedl, M.: Rule-based activity recognition framework: challenges, technique and learning. In: PervasiveHealth, pp. 1–7 (2009)
27. Lee, V.Y., Liu, Y., Zhang, X., Phua, C., Sim, K., Zhu, J., Biswas, J., Dong, J.S., Mokhtari, M.: ACARP: auto correct activity recognition rules using process analysis toolkit (PAT). In: Donnelly, M., Paggetti, C., Nugent, C., Mokhtari, M. (eds.) ICOST 2012. LNCS, vol. 7251, pp. 182–189. Springer, Heidelberg (2012)
28. Sama, M., Elbaum, S., Raimondi, F., Rosenblum, D.S., Wang, Z.: Context-aware adaptive applications: fault patterns and their automated identification. IEEE Trans. Softw. Eng. **36**, 644–661 (2010)