APPLYING MODEL CHECKING TO PERVASIVE COMPUTING SYSTEMS

LIU, YAN

(B.Eng., Southeast Univeristy (China), 2009)

A THESIS SUBMITTED FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF COMPUTING

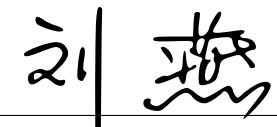
NATIONAL UNIVERSITY OF SINGAPORE

2014

Declaration

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.



LIU, Yan

11 June, 2014

Acknowledgements

Foremost, I would like to express my sincerest gratitude to my supervisor Dr. Jin Song Dong, for the continuous support throughout my Ph.D study with his patience, motivation and immense knowledge whilst allowing me the room to work in my own way. His guidance helped me in all the time of research and writing of this thesis. I attribute my PhD degree to his encouragement and effort and without him this thesis, too, would not have been completed or written. One simply could not wish for a better or friendlier supervisor.

My sincere thanks also goes to Prof. Jun Sun and Prof. Yang Liu who introduced me the powerful model checking technique and gave me solid training on applying this technique to real world problems. Their diligent working attitude and enthusiasm of advancing PAT model checker have influenced me during the entire PhD study. I have special thanks to Dr. Xian Zhang, Ling Shi and Lin Gui for their research collaboration.

In addition, I would like to thank Prof. Mounir Mokhtari and Dr. Jit Biswas for their support in the AMUPADH project and their valuable advice and insight. I also thank the rest of AMUPADH team members, especially Dr. Thibaut Tiberghien, Alwyn Lee Vwen Yen and Kenneth Lin Jin Hong for their help in explaining the system and providing important data sets for my experiments.

To my seniors, Dr. Chunqing Chen, Dr. Shaojie Zhang, Dr. Manchun Zheng and fellow

students Songzhen Song, Tian Huat Tan, Truong Khanh Nguyen, Shuang Liu, Guangdong Bai, Li Li, Manman Chen- thank you for all your support and friendships throughout my PhD study.

Lastly, I would like to thank my family for all their love and encouragement. For my parents Zhihua and Lingjuan who raised me full of love and supported me in all my pursuits. And most of all for my loving, encouraging and patient husband Junwei whose faithful support during the final stage of this Ph.D is so appreciated.

Contents

List of Tables **i**

List of Tables i

List of Figures **i**

List of Figures ii

1 Introduction **1**

1.1 Summary of This Thesis 3

1.1.1 Challenging Problems 3

1.1.2 Thesis Structure 4

1.2 Outline 6

1.3 Acknowledgement of Publications 7

2 Background **11**

2.1 Pervasive Computing Systems 11

2.1.1 The Typical Architecture 12

2.1.2 Important Features and Challenges 14

2.2 Model Checking 16

2.2.1 Basics of Model Checking 17

2.2.2 Concurrent System Model Checking 17

2.2.3 Probabilistic Model Checking for MDPs 23

2.2.4 Real-Time Model Checking 27

3 A Running Example: AMUPADH Healthcare System 31

3.1 Environment Data Acquisition 32

3.2 Context Processing and Reasoning 33

3.3 Adaptation: Reminder Service Rendering 36

4 A Formal Analysis Framework 39

4.1 A Modelling Framework for Pervasive Computing Systems 43

4.1.1 Modelling Environments 43

4.1.2 Modelling System Design 46

4.1.3 Compose a Complete Model 51

4.2 Properties of Pervasive Computing Systems 51

4.2.1 Desirable Properties 52

4.2.2 Testing Purposes 54

4.2.3 Bounded Liveness Properties 55

4.3 Related Work 57

5 Case Study: Formal Modelling and Verification of AMUPADH System 61

5.1 System Modelling 61

5.1.1 Environment Model 62

5.1.2 Sensor Model 63

5.1.3 Controller and Reasoning Engine Model 64

5.1.4 Reminding System Model 66

5.2 System Verification 66

5.2.1 Deadlock freeness (P1) 66

5.2.2 Guaranteed Reminders (P2) 68
5.2.3 Contradict Knowledge 72
5.2.4 Conflicting/False Reminders 72
5.3 Discovery of Unexpected Bugs 73
5.4 Discussion 75

6 Rule Anomalies Detection: A Model Checking Approach 79
6.1 ACARP Rule Checking System 81
6.1.1 Drools Rule Engine 82
6.1.2 System Workflow 82
6.1.3 Translating Drools rules to CSP# 83
6.1.4 Detection of Rules Anomalies 85
6.2 Experiments and Discussion 86
6.3 Related Work 88

7 A MDP-based Approach for Reliability Analysis 91
7.1 System Modelling using MDPs 94
7.2 Reliability Analysis Problems 97
7.2.1 Reliability Prediction 98
7.2.2 Reliability Distribution 99
7.2.3 Sensitivity Analysis 100
7.3 Analysing Reliability on AMUPADH system 100
7.3.1 System Modelling using MDP 103
7.3.2 Reliability Analysis Experiments 103
7.4 Related Work 106

8 Conclusions **109**

8.1 Summary 109

8.2 Future Challenges 111

8.2.1 Integrated Formal Modelling 111

8.2.2 Scalable Verification 113

9 Bibliography **115**

A Operational Semantics of CSP# **127**

B A Complete List of Rules **129**

C Case Study on A Transmission Protocol: CSMA/CD **133**

C.1 CSMA/CD: A Collision Detection Protocal for Local Area Network 134

C.2 Model for CSMA/CD protocol 135

C.3 Verification and Experimental Results 139

Summary

Pervasive computing systems are 'aware' of and self-adaptive to its environment changes. Many successes have been achieved in laboratories especially for activity monitoring. However, such systems are not widely deployed due to, not only scalability and a lack of guarantees for correctness and reliability, but also the fact that those systems are designed for demonstration purpose with well controlled scenarios in a specific lab environment. Existing approaches such as software testing and simulation are laborious and not sufficient since only partial system behaviours are explored. Formal methods, especially model checking techniques are needed to model and reason the real environment. In this thesis, we propose to apply model checking techniques to systematically analyse pervasive computing systems.

First, a formal modelling framework is proposed with general modelling patterns for both the system design such as concurrent communications, context reasoning behaviours etc. and the environment including the human behaviours. Critical requirements concerned by stakeholders are specified as assertions which are verifiable against the system model. Secondly, we present a systematic rule anomaly detection approach. A tool is developed to automatically translate Drools Rules to CSP# modelling languages. Rule anomalies can then be detected automatically by reusing existing verification algorithms. Furthermore, MDP-based probabilistic model checking techniques are applied to perform reliability analysis. We target at three questions: 1) reliability prediction- "What is the overall reliability of the system based on known component reliability?"; 2) reliability distribution- "To reach a certain overall system reliability, how reliable should the sensors/networks be?"; 3) sensitivity analysis- "Which node (could be a sensor or network device) has the most critical impact to the overall reliability?".

Last but not the least, case studies on a real-world pervasive computing system AMUPADH, demonstrate the usefulness of our approaches. AMUPADH is designed for monitoring and assisting elderly with dementia to live independently and is deployed in a Singapore based nursing home. Existing model checkers such as PAT and RaPid are adopted for carrying out verification experiments. Unexpected bugs and system flaws are exposed which are confirmed by system engineers.

Key words: **System Analysis, Model Checking, Pervasive Computing Systems, Ambient Assisted Living System, Healthcare, Correctness Analysis, Reliability Analysis, Case Study**

List of Tables

5.1 Experiment: Deadlock Freeness Checking 67

5.2 Experiment: Component Deadlock Freeness Checking 68

5.3 Experiment: Guaranteed Reminders Checking 72

5.4 Experiment: Testing Faults 74

6.1 Experiment: Anomaly Detection in Activity Recognition Rules 87

7.1 Experiment: Reliability Prediction 103

7.2 Experiment: Reliability Distribution 105

C.1 Components of CSMA/CD Model 137

C.2 Experiment: Verification of CAMS/CD Protocol 142

List of Figures

2.1 The Common Architecture of Pervasive Computing Systems 13

2.2 Architecture Design of PAT 18

3.1 AMUPADH: An Overview of the System 32

3.2 AMUPADH: Sensor Layout in the Bedroom 34

4.1 Formal Analysis Workflow 41

5.1 Model: Patient Behaviours 62

5.2 Model: Surrounding Environment Modelling 62

5.3 Model: Sensor Behaviours 63

5.4 Model: Reminding System Behaviours 66

6.1 Auto Correction System Component Diagram 83

7.1 Bathroom Scenario- TNO: Tap Not Off 96

7.2 Workflow: (a) reliability prediction; (b) reliability distribution; (c) sensitivity analysis 98

7.3 Bedroom Scenario- UWB: Using Wrong Bed 101

7.4 Experiment: UWB- Sensitivity Analysis on Nodes 106

C.1 Algorithm of CSMA/CD Protocol 135

C.2 Model: the *Sender* 138

C.3 Model: the *Bus* 138

C.4 Model: the *BroadcastCD process* 139

C.5 Model: the *CSMACD protocol* 139

C.6 Model: the *Collision detection in a given bounded delay* 141

Chapter 1

Introduction

Envisioned by Mark Weiser in the 90's, ubiquitous computing, aka pervasive computing is becoming the new computing paradigm of the 21st century, that computers disappear from the environment and "weaved themselves into the fabric of everyday life until they are indistinguishable from it" [84]. Significant hardware developments such as location sensors, wireless communication and mobile computing technologies have advanced Weiser's vision toward reality. Nowadays, the pervasive computing system (referred to 'PvC system' in the rest of the thesis) is emerging as a promising solution to problems risen with the proliferation of ageing population in all industrialised societies, e.g. creating enormous costs for the need of intensive care of elder 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 [56, 83, 63]. These systems usually incorporate complex technologies in a layered architecture design: a physical layer with sensors to monitor the environment and user behaviours; a middleware layer to manage and reason the sensed contexts so as to be aware of what's happening in the environment; an application services layer to make adaptations to environment changes by invoking actuators. Consequently, PvC systems are highly complex due to concurrent interactions among all these layers.

PvC systems can be safety critical, especially for smart healthcare systems built for elderly people. They are intelligently adaptive to the environment and fully automatic with little or no human supervision. Consequently, an error occurred in the system could harm the user's safety. For example, if a call-for-help reminder fails to be sent when the elderly user falls, he/she could be left unattended for a long time causing severe consequences. As a result, it is essential to adequately test and verify the system before they are deployed. However, these complex systems are developed without effective techniques to guarantee its correctness and reliability. Traditional techniques such as simulation and testing are evidenced to be expensive and not complete. In fact, to set up all hardware devices for testing is of high cost and time consuming. Furthermore, it is an impossible task for system engineers to consider all possible scenarios during development.

On the other hand, formal methods, especially model checking techniques are potential solutions to combat the weakness of these conventional methods. Model checking [37] is an automatic technique that can establish, via exhaustive analysis of the model of a system, whether its behaviour is correct with respect to a given specification. There are a number of successful stories in past years [86, 52]. Recently, it gained the most attentions for Intel's breakthrough on validating their new processor Core i7 fully by model checking and without using a single test case [38]. Model checking has a number of advantages compared with traditional techniques. It is automatic and complete with counterexamples generated to help the designers pinpoint the sources of the system flaws. Besides, model checking techniques do not require the actual system to be deployed and they can easily scale up to larger system models by proper abstraction and applying advanced state space reduction algorithms.

1.1 Summary of This Thesis

A number of formal approaches have been proposed for analysing PvC systems. However, most of them can only be applied to some component of the systems and some of them are not automatic which is infeasible for large systems. In this section, we summarise some of the challenging problems and propose our solutions to them.

1.1.1 Challenging Problems

- Correctness is essential to PvC systems which are usually fully automatic with little or even no human supervision. In order to apply model checking techniques, the foremost step is to properly model the system. A model of a PvC system should include models of all its components (i.e., sensors, network communication, rule-based reasoning and application adaptation) and the interactions among them. However, up till now, there is no modelling approaches to integrate all the component models in one framework. Besides, the system is usually user-centred. Modelling of the user behaviours is also important but often omitted in existing works.

- Rule-based activity recognition based on multi-modal sensor readings have been proposed to enable a PvC system to be shared by multiple users [82, 29, 74]. However, those rules are manually defined and error-prone. Incorrect or vague rules, could impair the system's capability in recognising activities, which further result in a lack of, or inappropriate service to be offered. Due to the relatively large number of rules and various scenarios that have to be tested in actual deployments, an automatic rule verification approach is needed while existing methods are not directly applicable. Most of the existing approaches were developed in 1990's for stateless rules (where knowledge are not shared during different runs of rule evaluation) while nowadays stateful rules are used more often in practical reasoning systems.

- A PvC system is considered reliable when all the assistance services are delivered at the right scenario to the right user. However, there are many causes leading to a unreliable system, e.g., sensors (less sensors or low-capability but cheaper sensors are used due to budget constraints [59, 65]) have limited detecting range and battery life, signal strength of wireless network drops with the increasing of distance. Thus, reliability analysis is critical for improving the quality of services provided by the system. Nevertheless, nondeterminism caused by unpredictable user behaviours prevents the direct use of existing techniques.

- There have been a number of smart systems being studied using formal methods in the past years. Those case studies are considered simple in the sense of limited concurrency. They are either single-user based or using simple sensors with less or no use of rules. However, smart systems nowadays are complex adopting a multi-modal sensor platform (also known as sensory data fusion) to enable multiple users sharing the environment concurrently. Thus, large case study on a complex real-life application is desired to show the feasibility of model checking technique.

1.1.2 Thesis Structure

To meet the challenges listed above, we propose a systematic formal analysis approach including specific methods targeting at different problems. In summary, the contribution of the thesis are explained below.

Correctness Analysis via Concurrent System Model Checking We propose a formal framework to systematically analyse PvC systems. Firstly, modelling patterns for unpredictable user behaviours and concurrent interactions between system components are proposed and illustrated with examples. Furthermore, we formally specified critical properties like safety (nothing *bad* happens) and liveness (something *good* eventually happens)

extracted from interviews of the stakeholders (system designers and caregivers of system users). Finally, we demonstrate a case study on a smart healthcare system for mild dementia patients, AMUPADH [11]. AMUPADH system has a multi-person sharing environment which exhibits additional complexity in terms of concurrent interactions. We adopt CSP# as the modelling language for its rich set of syntax in modelling concurrent system with hierarchies. Critical properties such as deadlock freeness and guaranteed reminder service are verified using Process Analysis Toolkit (PAT model checker) [77] (a self-contained framework for modelling, simulating and reasoning of concurrent and real-time systems). Multiple unexpected bugs such as conflicted reminders are detected at the early design stage.

Automatic Rule Anomaly Detection via Model Checking ACARP, is proposed as an automatic rules verification approach based on exhaustive manipulation of all possible scenarios. It is discovered that rule anomalies such as conflict rules can be represented using formal property specifications such as reachability and liveness properties. Thus,the rules verification problem can be transformed into a model checking problem. By adopting the formal modelling framework proposed in the first work, a scenario model is constructed. Further, ACARP automates the rule modelling by translating rules into CSP# modelling language for its support of external method calls. In such a way, the rule anomalies can be automatically detected by reusing the existing model checking algorithms efficiently. Experiment results show its usefulness to detect non-reachable, redundant and conflict rules.

Reliability Analysis via MDP-based Probabilistic Model Checking Markov Decision Process (MDP) is chosen as the modelling formalism for its support of modelling both probabilistic and non-deterministic choices. Based on the MDP models, three general questions of interest to end-users and developers are investigated, i.e., 1) *Reliability prediction*, "what is the overall system reliability if reliability of all its components and subsystems

are known, considering all possible user behaviours, and unreliable factors?"; 2) *Reliability distribution*, "what is the reliability required on subsystems or some devices if there is an expected reliability on overall system?", which provides decision support for cost-effective selections of software or hardware components; 3) *Sensitivity analysis*, which is important to found out the most critical parts to system reliability, based on quantitative measurement, that relatively more efforts and fund can be spent on. A case study on the reminding system of AMUPADH is demonstrated showing the system reliability is below 50%. Experiments also suggest that increasing the reliability of Wi-Fi network is more effective to improve the system reliability than replacing certain sensors.

1.2 Outline

In this section, we briefly present the outline of the thesis and overview of each chapter. Chapter 2 investigates the background information about common architectures and features of PvC systems. It also introduces the model checking techniques developed for concurrent, real-time and probabilistic systems respectively.

Chapter 3 illustrates a typical PvC system, AMUPADH which serves as a running example in this thesis. AMUPADH system is a smart healthcare for assisting independent living of elderly dementia people. The architecture includes three layers, i.e., data acquisition by multiple sensors on physical layer, context processing and reasoning on middleware layer and reminder service rendering on application layer.

Chapter 4 - 7 are the main chapters of this thesis and have the following structure. Following an introduction of the specific problem and its challenges, we demonstrate our proposed solution in details. A case study will be used to show the usefulness of our approach. Each chapter will be closed with a discussion of related works.

Chapter 4 presents the formal modelling framework with modelling patterns for the im-

portant components of PvC systems. We also identify the critical requirements commonly asked by stakeholders of smart systems.

Chapter 5 applies our formal analysis approach to AMUPADH system. A number of properties are verified against the system model which includes deadlock freeness, guaranteed reminders etc..

Chapter 6 further extends our formal analysis approach to rules verification. Considering there are large number of rules which are being frequently changed, we design a tool for automatic rule modelling using a translation approach. By specify rule anomalies as formal logic expressions, we are able to detect the redundancy, and conflicts in the reasoning rules.

Chapter 7 investigates reliability analysis using MDP-based probabilistic model checking techniques. We explore three problems which are reliability prediction based on known reliability value of system components, reliability distribution on certain nodes upon a reliability requirement on the system and sensitivity analysis aiming to find the most critical component which affects the system reliability.

Finally, Chapter 8 concludes the thesis with a discussion on open problems.

1.3 Acknowledgement of Publications

Most of the the work presented in this thesis has been published or accepted in the international conference proceedings or journals.

- **ACARP: Auto Correct Activity Recognition Rules using Process Analysis Toolkit [42]** was published at the 10^{th} International Conference on Smart homes and Health Telematics (ICOST 2012). This work is presented in Chapter 6.

- **Formal Analysis of Pervasive Computing Systems [46]** was presented at the 17^{th} IEEE International Conference on Engineering of Complex Computer Systems (ICECCS 2012). This work is partially presented in Chapter 4 and Chapter 5.

- **Formal Modelling and Verification of Pervasive Computing Systems [44]** was presented at the 1^{st} French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013) as a PhD paper. This work reported partially the work in Chapter 4-7.

- **MDP-based Reliability Analysis of an Ambient Assisted Living System [45]** was accepted for presentation at the 19^{th} International Symposium of Formal Methods (FM 2014). This work is presented in Chapter 7.

- **Towards Formal Modelling and Verification of Pervasive Computing Systems [47]** was published on Transactions on Computational Collective Intelligence (TCCI) XVI, 2014. This work includes contents from Chapter 4-6.

Besides, I have also contributed in the following publications which are remotely related to but not part of this thesis.

- **Modelling and Verification of Transmission Protocols: A Case Study on CSMA/CD Protocol [72]** was presented at the 1^{st} International Workshop on Model Checking Secure and Reliable Systems (MoCSeRS 2010).

- **Improved BDD-based Discrete Analysis of Timed Systems [57]**. It is published in the 18^{th} International Symposium on Formal Methods (FM 2012).

- **Modeling and Verifying Hierarchical Real-time Systems using Stateful Timed CSP [76]**. It was published in the journal, ACM Transactions on Software Engineering and Methodology (TOSEM).

For all these publications, I have substantial contribution in both theory and implementation.

Chapter 2

Background

2.1 Pervasive Computing Systems

A PvC system is an intelligent system with sensors, network and software system. It is context-aware and automatically adaptable to environment changes such as turning on the light when a person enters the room. Moreover, such system is no longer a static piece of software, but a complex system build up upon technologies across multiple disciplines of computer science and engineering, including wireless sensor network, rule-based reasoning software, distributed computing, human computer interaction and service oriented architecture etc. This type of system first introduced by Mark Weiser [84] refers to the seamless integration of devices into the users everyday life. Appliances should vanish into the background to make the user and his tasks the central focus rather than computing devices and technical issues. PvC systems are gaining intense attention and are emerging. Projects such as Oxygen [54] in MIT, Aura [17] in Carnegie Mellon University and AMUPADH [11] in Singapore have been launched years ago to achieve an intelligent world.

A PvC system is also referred to a number of other names. For example, "Ubiquitous Com-

puting", first named by Mark Weiser and his colleges in Xerox PARC. IBM related this notion to the slogan of "everywhere at anytime" as this technology should be pervasive. However, Phillips likely to call it Ambient Intelligent System and "Ambient Assistive Living" is usually adopted in healthcare domain. Researchers from UK are more familiar with the name "Sentient Computing". Other names such as "Context-Aware System", "Smart System" and "Calm Technology" are also used by researchers and industry people. However, all the names refer to the similar technology which is pervasive and intelligent. The difference between these names are purely academic said in the paper [67].

In this section, the necessary background of PvC systems is introduced. It includes the most common architecture of the system and the important features and challenges of the system for the research purposes of this thesis.

2.1.1 The Typical Architecture

Many approaches for implementing PvC systems are proposed in the literature [10]. These approaches differ due to special requirements and conditions such as location of sensors, the amount of possible users or the available resources of the used devices etc.. When analysing the various design approaches in modern applications, a common architecture is identifiable. The system usually adopts a layered design, as shown in Figure 2.1.

The first layer consists of a collection of different sensors. It is notable that the word 'sensor' not only refers to sensing hardware but also to every data source which may provide usable context information. This layer seamlessly monitors the changes in the environment such as changing of temperature, user's presence/ absence. It also provides raw contexts to other layers.

These raw contexts are then aggregated at the second layer, the middleware. The middleware layer is usually implemented on a centralised server for the purpose of context

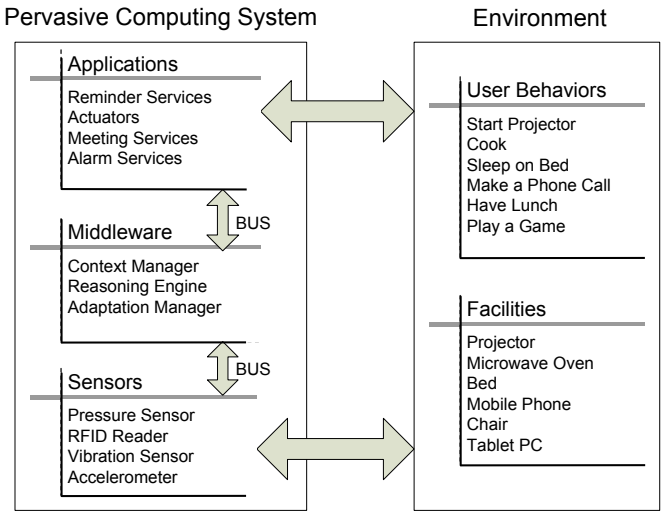


Figure 2.1: The Common Architecture of Pervasive Computing Systems

management and reasoning. Raw context information is interpreted from sensing signals to software understandable contexts. For example, a '1' in a message from the shake sensor on shower pipe is interpreted to be a context 'Showering' and some id '1' in a message from RFID reader in the shower room is known as 'user Jane is in the shower room'. Contexts interpreted from sensors are known as low-level contexts. The reasoning process in the middleware further combines these low-level contexts to infer high level contexts. For example, combining the contexts 'Showering' and 'user Jane is in the shower room', a high-level context information is inferred as 'user Jane is taking shower'.

In the third layer, the application layer, the responsive reactions such as prompting reminder services are implemented. Based on various high-level contexts provided from the second layer, different pre-defined services are activated for specific scenarios. For instance, in a smart meeting room system, the meeting service will automatically turn on the projector when a meeting scenario is detected.

The system interacts with the environment via sensor layer to detect changes and application layer to feedback/ adapt to changes. The architecture of PvC systems has evolved to be

layered during the last years. This separation of detecting, processing and using contexts is necessary in order to improve extensibility and reusability of systems. Understanding of the common architecture is critical to analyse PvC systems. Based on the architecture, it is able to identify the important components and their functions, find out the connections between different layers and more importantly to extract information/ knowledge that are shared among components.

2.1.2 Important Features and Challenges

PvC systems are complex that they usually incorporate many different technologies. They are systems consisting of heterogeneous sub-systems yet different from any existing computer systems [68, 71]. From the literature survey and the research experience, some of their distinct features to the interest of this thesis are summarised as follows.

Heterogeneity The innovation of the PvC system is that it integrates all the possible technologies developed so far to make computing devices serve people quietly. Then the challenge is how to integrate all the heterogeneous technologies in to one framework and make them working together smoothly. This heterogeneity exists in all the necessary parts of the system [68]. Sensors are the most obvious example. There are enormous sensors developed in different companies and countries and based on different standards [23]. The system usually relies on multi-modality of these sensors which includes sensors that detect pressure on the bed, sensors that could sense the door open/ close and sensors which monitors the water flow of the shower pipe. Furthermore, the sensors usually have very different refreshing rates. For instance, a pressure sensor may send the data every 1 millisecond while a reed switch sensor on the door may send its data every 1 second. These different refreshing rates often cause the reasoning engine to conclude false results due to the incomplete knowledge of the environment. Moreover, this heterogeneity could be even a nightmare for

engineers when the system is deployed in the real environments. A large team of engineers where each one of the them is specialised in a particular technology is needed to cooperate in the deployment. A failure happened in the system may involve multiple engineers working together to pinpoint the source of the problem.

Thus, the heterogeneity becomes a challenge to analyse PvC systems. The analysis approach should be able to model these inherently different devices and components. Many existing technologies such as testing and simulation get hindered by this challenge. What's more, sensors and wireless networks exhibits unreliable behaviours, e.g., sensors fail from time to time and networks get congested causing message loss. Managing the system at an acceptable reliability rate is another key challenge.

Context-Aware and Adaptive Contexts are referred to any information that could be used in the system, especially the information of the environment such as the time, the temperature or the location where the person presents. PvC systems are aware of its contexts and continuously adapting to the context changes [71]. These two characteristics differentiate PvC systems from other computing systems. The context-awareness is accomplished by the context manager and the reasoning engine in the middleware. As introduced in Section 2.1.1, the context manager collects and interprets the context stream sent by the sensors and the reasoning engine combines and infers high-level, software-understandable contexts. Adaptations to the environment changes are made based on predefined rules. These important features further require the analysis approach be able to modelling the shared information and the concurrent communications among components.

System of Systems The emerging system-of-systems (SoS) concept refers to a collection of many independent, self-contained systems where their integration offers more functionality and performance than simply the sum of these sub-systems. In a PvC system, the

networked sensors, the business rules engines and various applications such as Bluetooth speaker, TV and so forth are likewise systems. Thus, a pervasive computer system is an SoS because it is an assemblage of components that are individually regarded as systems. The PvC system also satisfies Maier's definition of SoS or "collaborative system" [50], that 'its components fulfilled valid purposes in their own right and continued to operate to fulfil those purposes if disassembled from the overall system, and the components systems are managed (at least in part) for their own purposes rather than the purposes of the whole'. This feature requires the formal approach to have the ability of modelling and reasoning the compositional relations between sub-systems. Besides, it is a big challenge to analyse huge system state space yielded from the composition of sub-system state spaces.

2.2 Model Checking

Principal techniques for formally analysing complex system behaviours include testing, simulation, deductive reasoning, and model checking. *Simulation* and *testing* approaches test system outputs with certain inputs against the expected results. However, they are very expensive and infeasible for complex systems with various unexpected behaviours and not complete because only a subset of possible behaviours are covered. *Deductive verification* uses axioms and proof rules to prove the correctness of the systems, which can handle infinite state systems but it is a manual approach that is time consuming and requires expertise. *Model checking* is an automatic approach for verifying finite state systems. It differs from other methods in two crucial aspects: 1) it does not aim of being fully general; 2) it is fully algorithmic and of low computational complexity[1].

[1]The complexity of most model checking algorithms is proportional to the state space or the product of the state space and property.

2.2.1 Basics of Model Checking

Model checking [37] is a verification technique that explores all possible system states in a brute-force manner. In order to formally analyse system behaviours using model checking, a system model needs to be constructed and abstracted from the original system design using certain formalism accepted by a model checker. Hereafter, the requirements of the systems are specified as properties in proper logics. One common example is temporal logic, which can assert how the behaviour of the system evolves over time. Finally, the verification of the specification against the system model is then conducted automatically by a model checker. The result will be returned with witness traces or counterexamples. The analysing of the error trace may require modifications to the model and repeat the model checking process.

Since model checking needs to fully explore the system space, thus it is required for the system model to be finite with bounded data size and finite number of processes. However, to examine the large state space using limited processors and memories remains a big challenge in model checking domain. State-of-the-art model checkers based on explicit state-space enumeration can handle about 10^8 to 10^9 states. Thus, the main bottleneck of model checking techniques is the infamous problem, *state space explosion*. This is also a big problem in formally analyzing pervasive computing systems due to the various communications between system components and the shared data variables which explodes the system state space exponentially.

2.2.2 Concurrent System Model Checking

Our home grown model checker, Process Analysis Toolkit (PAT) [2] is a self-contained toolkit to analyze concurrent systems, which supports of system modeling using CSP#, animated simulation and automatic verification of properties specified in LTL semantics and others.

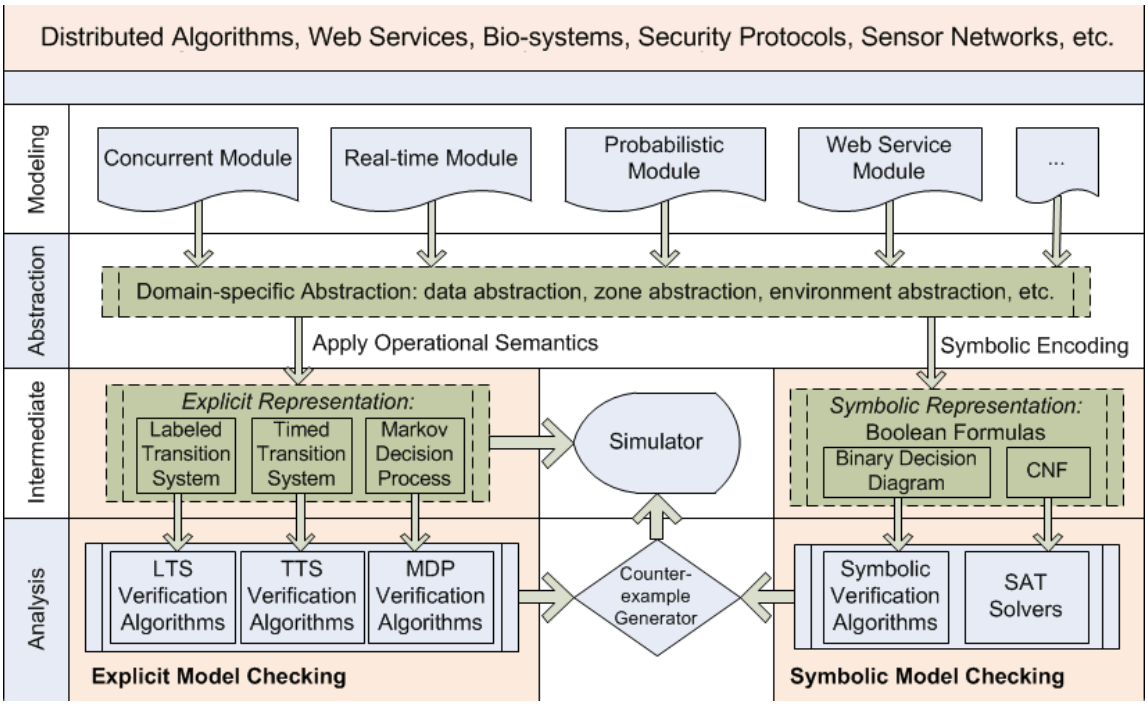


Figure 2.2: Architecture Design of PAT

Fig. 2.2 shows the architecture design of PAT with four components, namely the editor, the parser, the simulator and verifier. The editor is featured with powerful text editing, syntax highlighting and multi-documents environment. The parser compiles the system models and the properties into internal representation which is a labeled transition system. PAT adopts a fully automated abstraction technique to build an abstract finite state machine from the model. Further, it weakly bi-simulates the concrete model and, therefore, we may perform sound and complete LTL-X (i.e. LTL without the next operator) model checking or refinement checking upon the abstraction. The simulator allows users to perform various simulation tasks on the models: complete states generation of execution graph, automatic simulation, user interactive simulation, trace replay and etc. Most importantly, PAT implements several verification algorithms catering for safety, reach-ability, liveness properties verification, refinement checking and etc. To achieve good performance, advanced optimisation techniques are implemented, e.g., partial order reduction, process counter abstraction,

parallel model checking, etc. These verification algorithms perform on-the-fly exploration of the state space. If any counterexample is identified during the exploration, then it can be animated in the simulator for the purpose of debugging.

Modeling Languages- CSP# Modelling languages such as CSP [34] use mathematical objects as abstractions to represent systems and processes. System behaviours are described as process expressions combined with compositional operators, which are associated with elegant algebraic laws for system analysis. Nonetheless, modelling systems with non-trivial data and functional aspects using CSP remains difficult. Solutions are proposed such as new languages by integrating process algebras like CSP, CCS [53] with state based languages like Z languages or Object-Z language. However, declarative languages such as Z are very expressive but not executable. Automatically analysing system behaviours using these languages is extremely difficult. CSP# (short for communicating sequential programs) proposed in [75] instead extends CSP directly with low level programs eases the modelling and verification of computing systems. This language maximally preserves the original CSP and treats sequential programs as atomic events.

Sequential Programs as Events Shared variables offer an alternative means of communication among processes. They record the global state and make the information available to all processes.

1. **#define** *NoOfPatient* 2;
2. **enum** {*SILENT*, *FIRING*};
3. **var** *sensors*[9];

where *define*, *enum* and *var* are reserved keywords. The former defines a global constant, e.g., *NoOfPatient* which denotes the number of patients sharing the room. The middle one is a syntax sugar for *define* that enumerates the global constants. Silent and firing are

two statuses of the PIR sensor used to capture the human presence. The latter defines a variable, e.g., *sensors*[9] which stores the current status of each sensor. CSP# has a weak type system and therefore type information is not necessary for variable declaration. By default, all the above defined are treated as integers or arrays of integers. We use PAT to verify that the constraints hold given any system behaviour.

Composing Programs CSP# reuses high level composition operators in CSP since they are very useful in modelling system behaviours. Furthermore, process equivalence can be proved or disproved by appealing to algebraic laws which are defined for the operators.

A CSP# specification may contain multiple process definitions. A process definition gives a process expression a name, which can be referenced in process expressions. The following is a BNF description of the process expression

$$\begin{aligned} P ::= & \mathit{Stop} \mid \mathit{Skip} \mid e\{\mathit{prog}\} \rightarrow P \mid \mathit{ch}!\mathit{exp} \rightarrow P \mid \mathit{ch}?x \rightarrow P \mid \\ & P \setminus X \mid P;\ Q \mid P\ []\ Q \mid P \sqcap Q \mid \mathit{if}\ b\ \{P\}\ \mathit{else}\ \{Q\} \mid \\ & [b]P \mid P \| Q \mid P \| \mid Q \mid P \triangle Q \mid \mathit{ref}(Q) \end{aligned}$$

where P, Q are processes, e is a name representing an event with an optional sequential program *prog*, X is a set of event names (e.g., $\{e_1, e_2\}$), b is a Boolean expression, *ch* is a channel, *exp* is an expression, and x is a variable.

Stop is the process that does nothing. $\mathit{Skip} = \checkmark \rightarrow \mathit{Stop}$, where $\checkmark$ is the special event of termination. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. If e is attached with a program, the program is executed atomically together with the occurrence of the event. Channel communications, $\mathit{ch}!\mathit{exp} \rightarrow P$ and $\mathit{ch}?x \rightarrow P$ are considered as events. The former evaluates the *exp* (with the current valuation of the variables) and puts the value into the tail of the respective buffer in there is any and behaves as P. Process $\mathit{ch}?x \rightarrow P$ gets the top element in the respective buffer, assigns it to variable x and then behaves

as P. If the channel is declared without a buffer than the sending and receiving events have to be synchronized. Process $P \setminus X$ hides all occurrences of events in X. Sequential composition, $P;\ Q$, behaves as P until its termination and then behaves as Q. External choice $P \Box Q$ is solved only by the occurrence of a visible event. On the contrast, internal choice $P \sqcap Q$ is solved non-deterministically. Conditional choice *if* b $\{P\}$ *else* $\{Q\}$ behaves as P if b evaluates to true, and behaves as Q otherwise. Process $[b]P$ waits until condition b becomes true and then behaves as P. Notice that it is different from *if* b $\{P\}$ *else* $\{Q\}$. One distinguishing feature of CSP is alphabetized multi-processes parallel composition. Let P's alphabet, written as αP, be the events in P excluding the special invisible event τ. Process $P \| Q$ synchronizes common events in the alphabets of P and Q. In contrast, process $P \| \ | \ Q$ runs all processes independently (except for communication through shared variables). Process $P \triangle Q$ behaves as P until the first occurrence of an visible event from Q. A process expression may be given a name for referencing. Recursion is supported by process referencing. The operational semantics for CSP# is attach in Appendix A.

Property Specification

PAT supports a rich family of property specification which includes safety, liveness properties. These properties are supported in two ways regarding direct support with keywords and manually specifying using formulae of linear temporal logic.

Safety Property A safety property refers to "something bad never happens". In general, safety property requires the absence of deadlocks and similar critical states that cause the system to crash.

Deadlock-free is defined as the system will never enter a deadlock state which has no outgoing transitions. Deadlock is highly undesirable and mostly caused by a design error in concurrent systems. A typical deadlock scenario occurs when components mutually wait

for each other to progress. Deadlock freeness checking is directly supported in PAT using the keyword *deadlockfree*.

A more general form of safety property can be stated as a logic formula of the atomic propositions, e.g., $\neg(personInBedroom \wedge PersonInShowerRoom)$ is a safety property meaning that a person cannot be in two places at the same time.

Liveness Properties and Linear Temporal Logics Liveness properties mean that "something good will eventually happen". This property is useful in expressing the desirable system behaviors such as if the system can fulfill its mission. For example, PvC systems for healthcare are required to provide effective help when patients are in danger. This property can be expresses as system will eventually prompt a reminder to ask the patient to sleep when he is sitting on bed for too long.

Model checking based on temporal logic formulae has been proved effective as well as intuitive. For explicit state and event based modeling languages such as CSP#, state-based temporal logic such as Linear Temporal Logic (LTL) is a natural candidate for property specification and verification. In the following, an interpretation of LTL based on CSP# semantics is listed that this interpretation allows us to apply automata-based model checking of temporal logic formulae constituted with both event and state propositions. Let Pr be a set of propositions (formulated using predicates on global variables in CSP#). An extended LTL formula is defined as follows.

$$\phi ::= p \mid a \mid \neg\,\phi \mid \phi \wedge \psi \mid X\phi \mid \Box\phi \mid \Diamond\phi \mid \phi U \psi$$

where p ranges over Pr and a ranges over Σ. Let $\pi = \langle s_0, e_0, s_1, e_1, \cdots, e_i, s_i, \cdots\rangle$ be an infinite execution. Let π^i be the suffix of π starting from s_i.

$$
\begin{array}{ll}
\pi^i \vDash p & \Leftrightarrow s_i \vDash p \\
\pi^i \vDash a & \Leftrightarrow e_{i-1} = a \\
\pi^i \vDash \neg\, \phi & \Leftrightarrow \neg(\pi^i \vDash \phi) \\
\pi^i \vDash \phi \wedge \psi & \Leftrightarrow \pi^i \vDash \phi \wedge \pi^i \vDash \psi \\
\pi^i \vDash X \wedge \phi & \Leftrightarrow \pi^{i+1} \vDash \phi \\
\pi^i \vDash \Box\phi & \Leftrightarrow \forall\, j \geq i \bullet \pi^j \vDash \phi \\
\pi^i \vDash \Diamond\phi & \Leftrightarrow \exists\, j \geq i \bullet \pi^j \vDash \phi \\
\pi^i \vDash \phi U \psi & \Leftrightarrow \exists\, j \geq i \bullet \pi^j \vDash \psi \wedge \forall\, k \mid i \leq k \leq j-1 \bullet \pi^j \vDash \phi
\end{array}
$$

The simplicity of writing formulae concerning events is not purely a matter of aesthetics. It may yield gains in time and space. A model satisfies ϕ if and only if every infinite execution of L_P^V satisfies ϕ. This above example of liveness property can then be expressed as $PatientSitOnBedTooLong \rightarrow \Diamond Reminder\_Sleep$.

2.2.3 Probabilistic Model Checking for MDPs

Markove Decision Processes (MDPs) are standard models for stochastic optimisation and for modelling systems with probabilistic and nondeterministic or controlled behaviours [64, 80]. Verification algorithms designed for MDPs models are able to determine certain probabilistic behaviours of the system such as predicting probability of reaching a goal state from an initial state.

Markov Decision Processes

Discrete Time Markov Chains (DTMCs) and Markov Decision Processes (MDPs) are popular choices to model probabilistic systems. Given a set of states S, a distribution is a function $\mu : S \rightarrow [0, 1]$ such that $\Sigma_{s \in S}\, \mu(s) = 1$. Let $Distr(S)$ be the set of all distributions over S.

Definition 1 *A DTMC is a tuple $\mathcal{D} = (S, init, Pr)$ where S is a set of states; $init \in S$ is the initial state; $Pr : S \rightarrow Distr(S)$ is a transition function.* □

DTMCs are discrete stochastic processes satisfying the Markov property. A DTMC model can be expressed by a stochastic matrix $P : S \times S \to [0, 1]$ such that $\sum_{s' \in S} P(s, s') = 1$. An element $P(s_i, s_j)$ represents the transition probability from state s_i to state s_j. A state is an absorbing state if it has only self-looping outgoing transitions, i.e., $P(s_i, s_i) = 1$.

Definition 2 *An MDP is a tuple $\mathcal{M} = (S, init, Act, Pr)$ where S is a set of states; $init \in S$ is the initial state; Act is an alphabet; and $Pr : S \times Act \to Distr(S)$ is a labeled transition relation.* □

Different from a DTMC, there may be multiple distributions from a state, and each is labeled with a different action in an MDP. Intuitively, given a state s, an action (and the corresponding distribution) is first selected nondeterministically by a *scheduler*, and then one of the successor states is reached according to the probability distribution. A scheduler is a function deciding which action to choose based on the execution history. A DTMC can be defined by an MDP $\mathcal{M}$ and a scheduler δ, which we denote as $\mathcal{M}_\delta$.

With different schedulers, a state s may be reached with different probabilities. The measurement of interest is thus the maximum and minimum reachability probabilities. Let B be a set of target states. The maximum probability of reaching any state in B is denoted as $P^{max}(\mathcal{M} \models \diamond B)$, which is defined as: $P^{max}(\mathcal{M} \models \diamond B) = \sup_\delta P(\mathcal{M}_\delta \models \diamond B)$. Similarly, the minimum is defined as: $P^{min}(\mathcal{M} \models \diamond B) = \inf_\delta P(\mathcal{M}_\delta \models \diamond B)$ which yields the lower bound of the probability of reaching B. The supremum/infimum ranges over all, potentially infinitely many, schedulers. Existence of optimal memoryless schedulers, in which the decision for choosing next action/distribution based on the current state is independent of the previous choices, has been proved in [9]. Based on the result, different methods have been developed to calculate the maximum and minimum reachability probabilities. In this proposal and the RaPiD [31] model checker we adopted, the popular method, value iteration [9] is adopted.

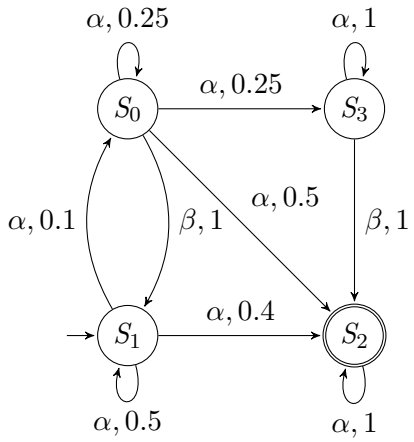
Value Iteration

Value iteration is an iterative approximation technique used to calculate the maximum and minimum probabilities of reachability, and often yields better performance than solving linear programs in practice [39]. In the following, we will demonstrate the application of value iteration on finding the maximum probability for reaching any state in B from the initial state. Let V be a vector such that, given a state s, $V(s) = P^{max}(\mathcal{M} \models \diamond B)$ is the maximum probability of reaching B from s. For instance, $V(init)$ is the maximum probability of reaching B from the initial state. First, using backward reachability analysis, we can identify the set of states X which have non-zero probability of reaching B, i.e., B is reachable from any state in X. Next, we iteratively build an approximation of V based on the previous approximation. Let V^i be the i-th approximation. We define V^i such that $V^i(b) = 1$ for all $b \in B$ and any i; $V^i(n) = 0$ for all $n \notin X$ and any i; and for each state $s \in X - B$, we have

$V^0(s) = 0$;
$V^{i+1}(s) = max\{\sum_{t \in S} Pr(s, a, t) \times V^i(t) \mid a \in Act(s)\}$.

The first equation defines the initial approximation. In the second equation, $Pr(s, a, t)$ is the probability of reaching state t from state s through action a. $V^{i+1}(s)$ is set to be the maximum probability of reaching B through any action based on the previous approximation. It can be shown that for every state s, $V^{i+1}(s) \geq V^i(s)$ and we can obtain V in the limit, i.e., $\lim_{i \to \infty} V^i = V$. In reality, it may take many iterations before V^i converges and thus value iteration is often stopped using a number of different conditions (e.g., when a fixed number of iterations have been reached or when the difference between two successive iterations falls below a certain threshold). Minimum probability of reaching B can be calculated similarly.

Example Consider an MDP below with the initial state S_0.



Let the target state set B be $\{S_2\}$. Note that all states have non-zero probability of reaching B. Each transition is labeled with an action and a probability. Transitions labeled with the same action belong to the same distribution. At state S_0, there is a nondeterministic choice between actions a and b. A scheduler decides whether to select a or b. Applying value iteration, we have $V^i(S_2) = 1$ for any i and

$$V^{i+1}(S_3) = max\{\, V^i(S_3), 1\} = 1$$
$$V^{i+1}(S_1) = 0.1\, V^i(S_0) + 0.5\, V^i(S_1) + 0.4$$
$$V^{i+1}(S_0) = max\{0.25\, V^i(S_0) + 0.5 + 0.25\, V^i(S_3),\, V^i(S_1)\}$$

It is then easy to get $V^0 = (0, 0, 1, 0)$; $V^1 = (0.5, 0.4, 1, 1)$; $V^2 = (0.875, 0.65, 1, 1)$; $V^3 = (0.96875, 0.8125, 1, 1)$; etc. □

Each iteration involves a matrix-vector multiplication, which has a complexity of $\mathcal{O}(n^2 \times m)$ in the worst case, where n is the number of states in S and m is the maximum number of actions from a state. Note that for sparse MDP models, the complexity is often $\mathcal{O}(n \times m)$. The number of iterations required to achieve certain numerical precision is related to the subdominant eigenvalue of the transition matrix [73].

2.2.4 Real-Time Model Checking

The modeling language Stateful Timed CSP# models real time systems with a comparison to Timed Automata and concept of timed refinement checking for verification of critical properties.

Language Syntax of *Stateful Timed CSP#*

The Timed CSP# modeling language is a timed extension of Communication Sequence Process (CSP) [34], its grammar is defined as follows.

Definition 3 (Process) *A* timed process *is defined by the following grammar.*

$$
\begin{array}{ll}
P = Stop \mid Skip & \text{– primitives} \\
\mid\ e \rightarrow P & \text{– event prefixing} \\
\mid\ [b]P & \text{– state guard} \\
\mid\ if\ b\ then\ P\ else\ Q & \text{– if-then-else} \\
\mid\ P \Box Q & \text{– general choice} \\
\mid\ P \| Q & \text{– parallel composition} \\
\mid\ P;\ Q & \text{– sequential composition} \\
\mid\ P \setminus X & \text{– hiding} \\
\mid\ P \mathrel{\widehat{=}} Q & \text{– process referencing} \\
\mid\ Wait[d] & \text{– delay} \\
\mid\ P\ timeout[d]\ Q & \text{– timeout} \\
\mid\ P\ interrupt[d]\ Q & \text{– timed interrupt} \\
\mid\ P\ within[d] & \text{– react within some time} \\
\mid\ P\ waituntil[d] & \text{– wait until} \\
\mid\ P\ deadline[d] & \text{– deadline}
\end{array}
$$

where P and Q range over processes, $e \in \Sigma$ is an observable event, b is a boolean expression, X is a set of event names and d is an integer constant.

Stop is the process does nothing but idling, also denotes deadlock. *Skip* states termination. Process $e \rightarrow P$ performs event e first and then behaves as P. Notice that e may be an

abstract event or a data operation, e.g. written in the form of $e\{x = 1;\ y = 2;\ \}$ or an external C# program. The data operation is used to update data variables and it is assumed to be executed atomically. A guard process is written as $[b]P$. If b is true, then it behaves as P, else it idles until b becomes true. A conditional choice is written as *if* b *then* P *else* Q. If b is true, then it behaves P, else it behaves Q. An unconditional choice is written as $P \Box Q$. The choice to choose which process to perform accords to what events are requested by its environment. Parallel composition is written as $P \| Q$, where P and Q may communicate via variables, or multi-party event synchronisation. Process $P;\ Q$ behaves as P until P terminates and then behaves as Q immediately. $P \setminus X$ hides occurrences of events in X by replacing them with τ (an unobservable event). Process $P \mathrel{\widehat{=}} Q$ defines P to be exactly as Q. Processes may communicate through message passing on channels. Channel buffer size must be greater or equal to 0. Notice that a channel with buffer size 0 sends/receives messages synchronously.

Timed process constructs can be used to capture common real-time system behaviour patterns. Process $Wait[d]$ delays the system execution for a period of d time units then it terminates. In process P *timeout*$[d]$ Q, the first observable event of P should occur before d time units elapse (since the process starts). Otherwise, Q takes control over after exactly d time units elapse. Process P *interrupt*$[d]$ Q behaves exactly as P (which may engage in multiple observable events) until d time units elapse, and then Q takes controls over. Process P *within*$[d]$ constrains that P must *react* (by engaging in an observable event) within d time units. Process P *waituntil*$[d]$ denotes P executes for at least d time units and process P *deadline*$[d]$ constrains P must terminate within d time units.

Compared to Timed CSP#, Timed Automata [5] which is popular for specifying real time systems during last decades, has certain deficiencies that it is not feasible in supporting compositional models. Timed Automata are powerful in designing real-time models with explicit clock variables. Real-time constraints are captured by explicitly setting/reseting

clock variables. A number of automatic verification supported for Timed Automata have proven to be successful (e.g. UPPAAL [41], KRONOS [12] and RED [81]). However, in industrial case studies of real-time system verification, system requirements are often structured into phases, which are then composed sequentially, in parallel and alternatively [32]. High-level requirements for real-time systems are often stated in terms of deadline, time out, and timed interrupt. Unlike Timed CSP#, Timed Automata lack high-level compositional patterns for hierarchical design. As a result, users often need to manually cast those terms into a set of clock variables with carefully calculated clock constraints. The process is tedious and error-prone.

Chapter 3

A Running Example: AMUPADH Healthcare System

AMUPADH is a project initiated in Singapore to design smart healthcare systems for monitoring and assisting elderly dementia people's daily living. 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.

The system developed in AMUPADH 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.). It 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 Zigbee network. The *controller* in the middleware interprets sensor signals into low-level

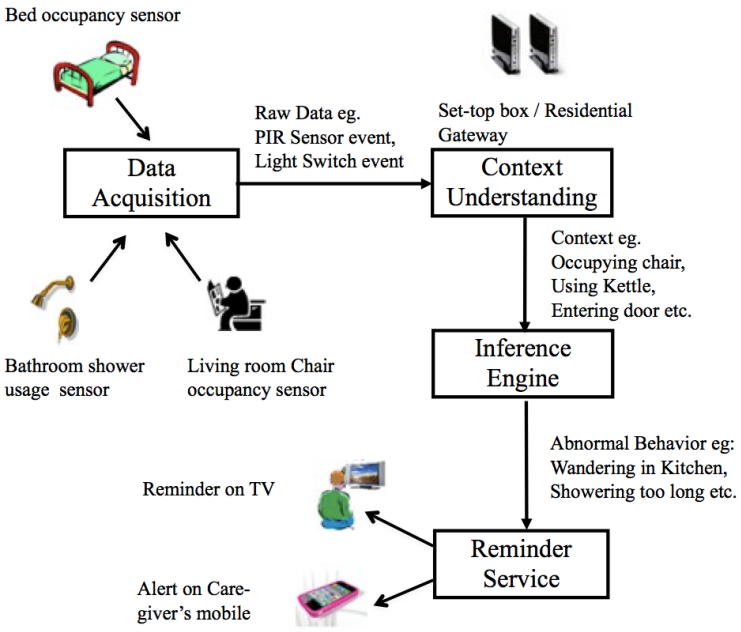


Figure 3.1: AMUPADH: An Overview of the System

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 recognised, the system will prompt a reminder requesting him to use his own bed.

3.1 Environment Data Acquisition

In the system, multiple sensors are deployed to acquire information from the home environment. For example, if someone turns on the shower tap, the shake sensor on shower pipe

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

will be triggered and change its status to *Unstationary*. A signal is generated and then sent to the central system via a Zigbee network. AMUPADH adopts a multi-modal-sensor design for user monitoring. This is due to users' privacy concerns, video cameras are refused in home environment.

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 3.2.

- **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.

3.2 Context Processing and Reasoning

Upon receiving a signal, the central system interprets it into low-level context, i.e., *sensor events* such as "Shower Tap On". Different low-level contexts are provided from different sensors. They are aggregated in the inference engine for reasoning and generating high-level con-

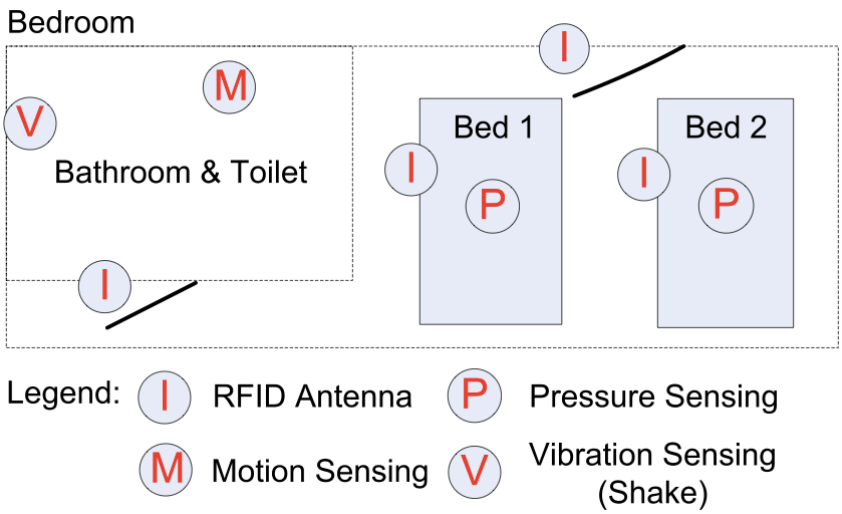


Figure 3.2: AMUPADH: Sensor Layout in the Bedroom

texts, *activities*. This task is performed by evaluating predefined activity recognition rules based on prior knowledge of user behaviours.

Rules are written in first order logic. A typical rule is like: if a shake sensor on shower pipe changes its status to *UNSTATIONARY* and lasts for 30 minutes and a PIR sensor captures movements of someone in the washroom, an abnormal behaviour, *showering for too long* is recognised. Then a message will be sent to the server indicating some person with a name is in an abnormal state of showering for too long. The messages are sent out via a shared bus within the central system. DroolsA typical rule are in the form of the following:

```
rule "Person is showering for too long"
  when
    Sensor ( id == "shakeShower",
        shakeState == Sensor.shake_state.UNSTATIONARY,
        durationInSecond >= 30 )
    Sensor ( id == "pirWashroom",
```

```
        pirState == Sensor.pir_state.FIRING )
    $p : Person( name != "nurse",
        location == Person.person_location.WASHROOM )
    $x : XMPPInterface()
  then
    $x.SendData( "ACTIVITY.error." + "ShowerTooLong"
        + "." + $p.getName() );
end
```

The condition of this rule uses two contexts from shake sensor on shower pipe and PIR sensor (for motion monitoring) in washroom and the context of user's location. This rule can be interpreted as: If the shake sensor on shower pipe changes to *UNSTATIONARY* and lasts for 30 minutes, PIR sensor captures movements of someone in the washroom, an abnormal behaviour, *showering for too long* is recognised. Then the engine will send a message to the server saying that some person with a name is in an abnormal state of showering for too long. The messages are sent out via a shared bus within the central system. The full set of 23 rules used in the system is listed in B.

Note that, AMUPADH aims for a multi-user sharing environment which is a challenging topic in the activity recognition area. In fact, it is not only important to know about which activity is being carried on but also who is doing this activity. This adds complexity to the process of defining rules and lowers the accuracy of activity recognition. Faults like forget to put person's identity into rule conditions could result in associating an wrong activity and subsequently wrong reactions to a wrong person.

3.3 Adaptation: Reminder Service Rendering

The reminding system listens to the messages sent from the inference engine and decodes it. A number of simple rules are defined to deal with reminder service rendering. For example, upon receiving the message $Activity.error.ShowerTooLong.personA$, the system will invoke the service of playing a preloaded sound reminder on bluetooth speaker located in the shower room correspondingly. In this case, the message is transferred via bluetooth technology. In general, different message transmitting technology are used for different rendering devices. For instance, for reminders on mobile phones, messages are transmitted from 3G network, while for iPAD case, the small home wide Wi-Fi network is used.

In AMUPADH, there are essentially six reminders provided for helping the patients in highly concerned situations.

- **UWB: Using Wrong Bed** Since a room in the RLA is shared by 2-3 people, the elder patient, especially new residence, tends to lie on a bed without recognising whether its his/her own bed.
- **SBTL: Sitting on Bed for Too Long** Some of the agitated patients often have sleeping problems. They are easily bothered and irritated by what is happening in the environment. A typical symptom is that the patient will get up at midnight and sit on the bed for very long time until assisted by nurses/caregivers.
- **SNS: Shower No Soap** Due to memory loss, dementia patients constantly forget the normal steps of performing daily activity. In the taking shower activity, the patient could forget what to do next right after the shower tap is turned on. It is reported by the nurses that some of the patient finish the shower very fast without applying soap. Concerned about the personal hygiene, patients presenting this behaviour need to be helped.

- **STL: Showering for Too Long** Similar to the SNS issue, some patients will stand under the shower head for a long time. It is a critical issue that exposing in the water for a long time could cause the patient black out. If not helped immediately, it will even cause death to the patient.

- **TNO: Tap Not Off** It is often the case that dementia patients forget to turn off the tap after showering. In order to save water and energy, this scenario is also detected and reminded in the system.

- **WiW: Wandering in Washroom** Caused by initiation problem, it is possible for the patient to forget at any step of the taking shower activity. Thus, a wandering behaviour is also typical and patients need to be assisted in such cases.

The smart home care system developed in AMUPADH project has been deployed in a Singapore based nursing home, PeaceHeaven[2] for a six-month real life testbed. This nursing home has 13 separate Resident Living Areas (RLAs), each designed as an individual home-like environment. The rooms are equipped with two/three beds with a shower facility. Three of these rooms each of which are shared by 2 or 3 people are selected for deployment.

[2]Located at 9 Upper Changi Road North, Singapore, 507706. Tel: +65-65465678.

Chapter 4

A Formal Analysis Framework

Pervasive computing systems aim to provide people with a more natural way to interact with information and services by embedding computation into the environment as unobtrusively as possible [84, 24]. They are fully automatic with little or even no human supervision. For example, systems are built for assisting the independent living of elderly with dementia where nurses or caregivers intervene only when receive call-for-help alerts. Thus, such systems are life-critical. It is highly important to know if the system behaves correctly as expected before the deployment of such systems.

Correctness analysis of these smart systems is a challenging task. Firstly, these systems are inherently complex. Revisited the general architecture in Chapter 2, the system usually adopts a layered design with sensors in the physical 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 PvC systems highly complicated [23]. Faults may appear in many situations with very different causes. For example, a false reminder that a reminder is

sent to the wrong user could be caused by the malfunctioning RFID reader or a wrongly defined reasoning rule; the conflict reminders could be caused by conflict rules or not well calculated timing issues. Furthermore, it is impossible to exhaustively enumerate all the possible scenarios. Various environment inputs and unpredictable user behaviours cause the system behaviours beyond control, especially when multiple users are interacting with the system simultaneously. Techniques like simulation and testing are time consuming and not sufficient. System engineers have to run the system multiple times only to find out the cause of an uncommon error. Even though, testing and simulation techniques can not provide enough guarantee of correctness since these technology can only explore partial system behaviours. From our experiences of working with system engineers, 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.

Formal methods especially model checking techniques are promising solutions that they do exhaustive state space analysis and can be applied in the early design stage of complex systems. By properly modelling the system behaviours and formally specifying the critical requirements, it is able to automatically verify important requirements by exhaustive search of all possible states of the system model. The violation of a requirement is usually witnessed by a counterexample (e.g. an execution trace from the initial state of the model) which provides a good guidance in pinpointing the source of the error. The first chanllenge to apply this technique is to properly model the system. Unfortunately, most existing models (e.g., TCOZ model in [21] and Ambient Calsulus model in [19]) have limited support for modelling hierarchical structures. Besides, no general patterns regarding the modelling of PvC systems are proposed in the literature. Henceforth, new methods are expected to be able to model the typical behaviours of the system such as concurrent communications,

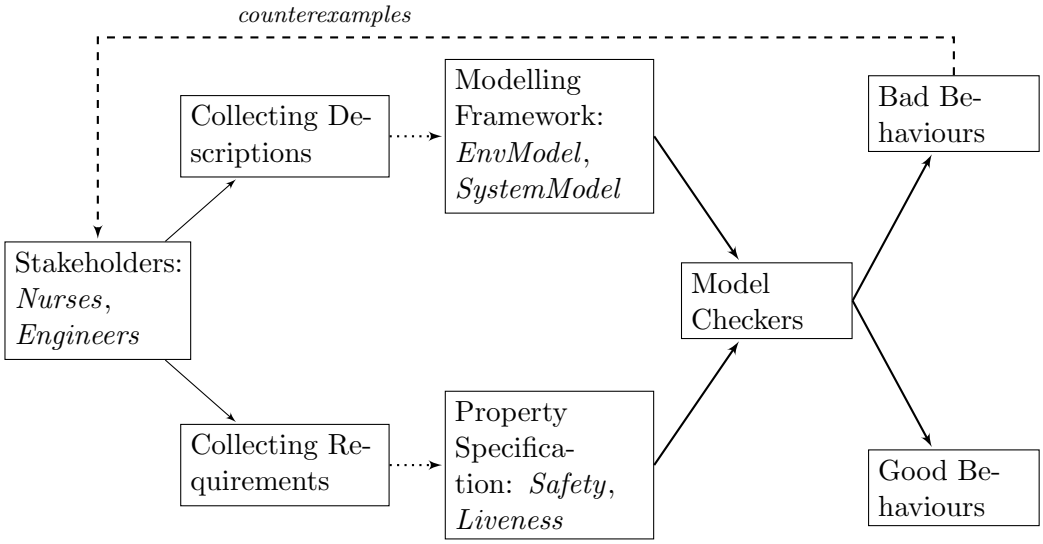


Figure 4.1: Formal Analysis Workflow

complex control logics and hierarchical structures. Furthermore, the constructed model should be supported by certain model checker such that automatic verification is possible for large systems.

To meet the challenges, we propose a formal modelling framework targeting at the general architecture of PvC systems. Based on the modelling framework, specifications of important correctness requirements are also provided. Thus, the typical workflow of our formal analysis approach is shown in Figure 4.1. Since PvC systems are user-centred, understanding and modelling of the user behaviours are important. Thus, the workflow starts with collecting user and system information from stakeholders (e.g., in AMUPADH, Nurses/Doctors/Engineers). In particular, we need to seek answers via observing the elderly people's behaviours and interviewing nurses and doctors for these questions: 1) What are the targeting activities of elderly people that are critical to be monitored? 2) How does the elderly people performing such activities? How to identify if the user is performing the activity abnormally that he/she needs help? 3) How does the nurses/doctors help the user in case of an abnormal behaviour identified? In addition, by interviewing system engineers and reading

design documents including pseudocode of the AMUPADH system, we need to know about 4) How does the system designed to identify the abnormal behaviour and provide assistance to the user? Meanwhile, the critical requirements such as the system will not be dead or reminders are always sent out at certain scenarios are also collected from both the nurses/-doctors and the system engineers. Next, with the knowledge of both the user behaviours and system design, we can formally model them as *environment model* and *system model* respectively. The separation of user model and system model enables the flexible reusing of them. The critical requirements are then formally specified as *properties*. Finally, a model checker is used to automatically verify properties against the system model. Counterexamples are usually generated as feedback to stakeholders for improving the system design. The contributions of our work are two-folds as explained below.

Firstly, we propose modelling patterns 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. We adopt CSP# [75] as the sample modelling language for its rich set of syntax in modelling concurrent system with hierarchies.

Secondly, we identify critical properties of PvC 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 PvC 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 [37]. Hence, design flaws can be detected at the

early design stage.

4.1 A Modelling Framework for Pervasive Computing Systems

PvC 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 introduced in Chapter 2.1.1. In this section, we discuss the important features of PvC systems layer-by-layer and propose corresponding modelling patterns for them. Besides, environment inputs perform an important role in PvC systems. Thus, along with the modelling of system components, we also propose modelling patterns for different environment aspects which are usually not included in most complex systems models.

4.1.1 Modelling Environments

PvC systems seamlessly interact with the environments and acquire context inputs from the users and objects like TVs and Beds. To some extent, PvC 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 analysing verification results.

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, 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 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 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 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 solutions are possible such as using a global variable or synchronous channels.

```
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 [75] 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.

4.1.2 Modelling System Design

PvC systems share the features such as layered architecture and concurrent communications. A common architecture of such systems is shown in Figure 2.1. 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 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 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 [58]. 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 Section 4.1.1. 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) [76] or Timed Automata (TA) [5]. 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 behaviours 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.

Modelling 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 modelling languages such as Probabilistic CSP (PCSP) [78] or Probabilistic Timed Automata (PTA) [39].

```
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 Figure 2.1, 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 Chapter 3.2 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 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) =
            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.

4.1.3 Compose a Complete Model

In PvC 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 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.

4.2 Properties of Pervasive Computing Systems

After system engineers finished the design of a PvC system, they are often asked to provide guarantees for correctness and even safety requirements. They may be asked to answer gen-

eral 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.

4.2.1 Desirable Properties

Properties regarding the good behaviours of the systems are desirable.

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.

Guaranteed Services

Well designed application services determine fundamental responsive behaviours of PvC 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,

```
#define PatientWandering (Pos_Person[1] == SHOWERROOM && WanderFlag);
#define LeaveRoomReminder (ReminderStage[WANDERING_IN_SHOWERROOM*2] ≠ 0);
□(PatientWandering → ◇ LeaveRoomReminder)
```

Here, □ and ◇ 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,

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

Model checking techniques for security problems are proposed in papers such as [51].

4.2.2 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 [22]. 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.

Properties in rules

Rule-based reasoning engines are popular in pervasive computing systems. The correctness of rules is essential to the correct behaviours 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.

4.2.3 Bounded Liveness Properties

Liveness properties properties such as guaranteed services checks if certain service is eventually delivered whenever an abnormal behaviour is detected. However, stakeholders often require more than "eventually prompted". The good behaviour should be whenever an abnormal behaviour is detected the reminder should be prompted within a certain amount of time. This is often referred as bounded liveness property which says something good will happen soon, in bounded time. It usually requires the properties to hold within a certain time bound. *Bounded liveness* property captures a wide range of timed requirements for safety critical complex systems. Taken AMUPADH system as an example, we propose some sample properties specifying timed requirements in the following. We choose the specification language which is supported in existing model checkers.

Guaranteed In-time Servicess

Taken example shown in 4.2.1. If the reminder is required to arrive within t time units once this wandering behaviour is detected, we need to extend the LTL property with real time constructs. Alternatively, we may use the timed extension of a branching time temporal logic (CTL [15]), TCTL [33] which is usually supported or partially supported in model checkers like UPPAAL. The guaranteed *Wandering_In_Showerroom* reminder prompted in time is then specified as:

$$\forall\ \Box\ (\texttt{PatientWandering} \rightarrow \forall\ \Diamond^{\leq \texttt{t}}\ \texttt{LeaveRoomReminder})$$

Guaranteed In-time Reminders with Probability

In the case that, the system is modelled with probabilistic behaviours, the bounded liveness will be changed to a liveness property bounded with both time and probability. For example, a requirement says the *Wandering_In_Showerroom* should arrive within t time units with a probability of 0.98 when the wandering behaviour is recognised. The property is then specified using PCTL, a probabilistic extension of CTL as follows:

$$\texttt{P}\ ^{\geq 0.98}\ \Box\ \texttt{(PatientWandering}\ \rightarrow\ \forall\ \Diamond^{\leq \texttt{t}}\ \texttt{LeaveRoomReminder)}$$

Bounded Message Delay

In network layer, message delay is a common problem due to network congestions. To resolving the congestions, networks follows the policy of collision avoidance and retry mechanism that nodes like sensor in the network will wait for a certain time period to retransmit its message. Thus, during the verification of network policies/ protocols, bounded message delay is a desirable property. This requirement can be specified similarly as **Guaranteed In-time Reminders**. An example is as follows:

$$\forall\ \Box\ \texttt{(Node.MessageSend}\ \rightarrow\ \forall\ \Diamond^{\leq \texttt{t}}\ \texttt{System.MessageReceived)}$$

Well Scheduled Reminders

One good behaviour of well scheduled reminders is that two different reminders are not prompted to the same user within a time period t. It is specified as follows:

```
WanderInShowerroom.PersonA --> ShowerNoSoap.PersonA && (C_WIS - C_SNS < t)
```

where C_WIS and C_SNS are clock readings of *Wandering_In_Showerroom* reminder and *Shower_No_Soap* reminder respectively. Expressing properties with clock values are possible in model checkers implemented with explicit clocks like UPPAAL.

Although formalisation of real-time requirements are strongly dependent on the specification logic supported by model checkers, the sample property specifications we listed here are common patterns and generally supported in popular model checkers like UPPAAL and PRISM. However, it remained a challenging and interesting task to verify properties with various timed requirements.

4.3 Related Work

Modeling Pervasive Computing Systems

TCOZ model of Smart Meeting Room TCOZ is essentially a blending of Object-Z with Timed CSP, for the most part preserving them as proper sub-language of the blended notation [49]. In [21], they manually modeled and verified the constraints and relations in the context aware systems with a case study of a smart meeting room system. The concurrent communicating patterns, sensor constraints and real time requirements are captured in their model. In order to reason about important system properties and keep the size of system model small, irrelevant implementation details such as how data transmits from sensor to system is abstracted away. This strategy is reasonable and effective. TCOZ is concise and yet powerful for modeling system behaviors, many features of context-aware systems can be naturally modeled using constructs of the language. However, the lack of automated reasoning tool support, properties against the system needs to be verified by manual proof/disproof. Thus it requires much expertise from the user which prevents this language to be adopted widely in real system modeling.

Advances are to be learned from TCOZ regarding the flexibility of modeling sensor constraints, sensor patterns and real time relations. However, our proposing approach differs with TCOZ in several aspects. First, we are aiming at proposing executable models that can be simulated and automatically verified. TCOZ however extends from the declarative languages are not automatically executable, thus hard to be supported in automated tools. Additionally, in order to better locate the errors in the system, counterexamples needs to be reported if a certain property is violated. This needs support from the language constructs how such as trace semantics defined in CSP# languages family. However, this is not well supported in TCOZ language.

Ambient Calculus model of Location-sensitive Smart Hospital [19] modeled another mobile application in a hospital which provides location sensitive service to guide the patient enter the proper room and prevent patient to get in dangerous places. They adopted the Ambient Calculus [13] for the modeling and verified certain properties against the critical requirements of the system. Locality in this modeling language is very well captured using ambients. However, only the patient's interaction with the environment is modeled in this case study. It is not clear of how other important features of PvC systems such as adaptation logics and sensor communications can be modeled. And again lack of automated reasoning tool adds more difficulty to this language to be applied in real scenario.

Verification of Pervasive Computing Systems

Fault Patterns and Automatic Identification of Context-Aware Adaptive Applications (CAAAs) Sama and Rosenblum et, al. in [70, 69] focused on analysing the adaptive behaviour which is essential PvC systems. They identified fault patterns of such systems based on the experience of designing electronic devices, proposed a innovative semantic model named Adaption Finite-State Machine (A-FSM) and proposed dedicated

verification algorithms to detecting those fault patterns based on the A-FSM. The fault patterns as well as the causes they identified is a good summary of existing PvC systems and the state matrix data structure further facilitates the efficient verification algorithm design.

Advanced techniques in their work such as A-FSM and state matrix data structure are intuitive for representing the context-aware systems. This can be learned to our proposing work by proper design. However, they didn't mention how to model a system in A-FSM or how models in other modeling language can be represented using A-FSM. In real cases, PvC systems have various architectures and system components. Profile which can be simply represented as states in A-FSM might be common in mobile applications but not in PvC systems for healthcare. Thus there should be a formally definition for a state in A-FSM in view of general PvC system. Moreover, they are able to identify fault patterns and perform static analysis of rules to check errors in rules. However, since their rules are priority based and the algorithms are also designed to tackle this feature, these fault patterns and algorithms may be not applicable for general rules based reasoning engines that doesn't have priority. For example, the deterministic property might be unrealistic in a smart system monitoring multiple persons because multiple rules can be triggered at the same time by different person performing different activities in different places. The system then diverges but this is not considered as an error.

Identifying and Formally Defining Properties in Verification of Pervasive Systems In [8], the authors proposed the key properties needs to be verified in PvC systems, which are the security, safety and usability issues. Different property specification languages are adopted for specifying different properties such as access control language RW for security of patient profiles, applied Pi-Calculus for un-traceability of RFID tags. and temporal logic for safety problems such as if a patient is in danger, assistance should arrive in given

time. However, they didn't discuss further on how these properties are to be checked against which kinds of models, since there is no such modelling language that can build a model checkable by all the mentioned properties in various specification logics.

Inconsistency Detection and Resolution Contexts gathered by sensors from the environment could be fault or inconsistent due to the limited accuracy of sensing technologies and sensor failures. And such inconsistencies could cause serious problems to the system such as fail to response to the environment change or adapt to false application. Wang and Gu et. al. [82, 29] proposed ontology based context reasoning which defines rules based on first order logic to specify the restrictions and consistencies among contexts. Xu and Cheung proposed dynamic inconsistency detection by a semantic matching and inconsistency triggering approach in [89]. They further proposed a formula based incremental checking approach based on a Consistency Computation Tree (CCT) [90]. However, these approaches highly reply on expert knowledge in identifying and specifying those consistency constraints as well as additional effort on developing consistency checking algorithms. In our approach, the only effort is to specify a general inconsistent scenario and model checking tools will automatically check all the state space to find such a scenario. In this work, we propose a formal modelling framework for PvC systems. Different modelling patterns are discussed according to the typical features of systems such as concurrent communications, context-awareness and layered architectures. We also provide environment modelling patterns which are usually not considered in modelling 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.

Chapter 5

Case Study: Formal Modelling and Verification of AMUPADH System

In this case study, we applied the proposed approach in Chapter 4 to analyse the concurrent behaviours[1] in AMUPADH system. 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.

5.1 System Modelling

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

[1]Real time and probability features are not included in this work. Modelling and verification with these two features are explored in Chapter 7 and Appendix C.

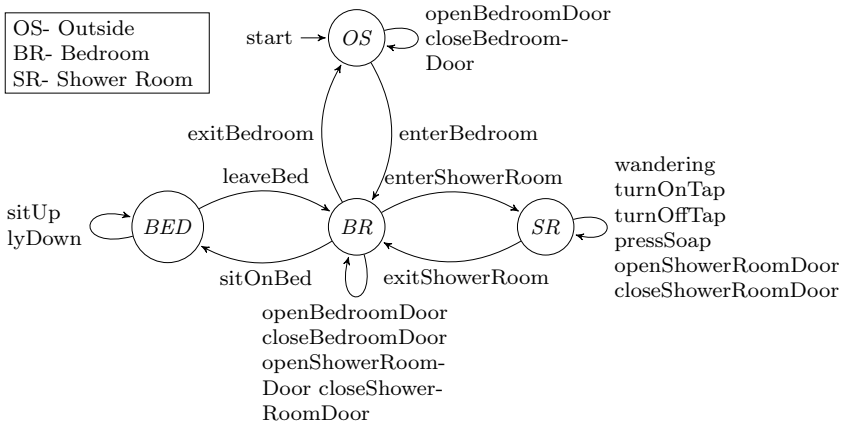
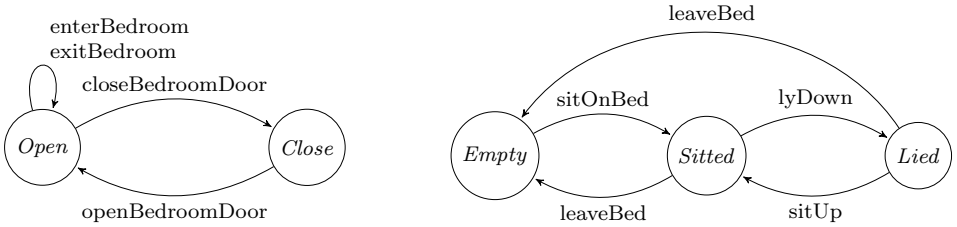


Figure 5.1: Model: Patient Behaviours



(a) Bedroom Door Behavior

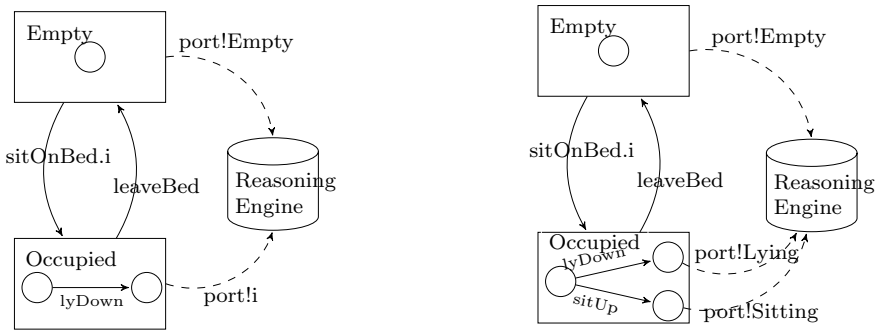
(b) Bed Behavior

Figure 5.2: Model: Surrounding Environment Modelling

5.1.1 Environment Model

As shown in Figure 5.1 and 5.2. These LTSs can be generated using simulation function of PAT. In Figure 5.1, 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 Figure 5.2a and 5.2b respectively.



(a) Bed RFID Reader (b) Bed Pressure Sensor

Figure 5.3: Model: Sensor Behaviours

5.1.2 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 5.3 shows the modelling of sensors using the bed RFID readers and bed pressure sensors as mentioned in Chapter 3.1. 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();
```

5.1.3 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 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()};
...
```

Syntax **if** (*cond*) $P1$ **else** $P2$ means if the condition *cond* is true then process $P1$ is executed, otherwise process $P2$ is executed. *ShakeTap* and, *SITTING*, *Error* and *WanderingInShowerroom* are constants which denote sensor ID, sensor status, command to reminding system (the other is *Normal*) and behaviour code. The rest of the rules are modelled similarly. The full set of rules used in AMUPADH is listed in Appendix B.

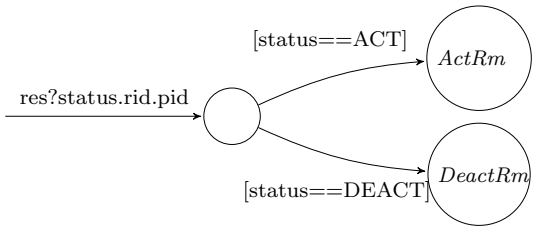


Figure 5.4: Model: Reminding System Behaviours

5.1.4 Reminding System Model

In the system, reminders are activated/ deactivated upon receiving corresponding messages from the controller. As shown in Figure 5.4, 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 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 introduced in Section 4.1.3.

5.2 System Verification

In this section, requirements concerned by system designers and users (patients/ nurses/ doctors) are formally specified and verified. Experiments are carried on a test bed- a PC with Intel Xeon CPU at 2.13GHz and 32GB RAM.

5.2.1 Deadlock freeness (P1)

Deadlock freeness property is directly supported in PAT using the keyword *deadlockfree*. However, there is a state space explosion problem of the *SmartRoom*() model which is cased

Property	Result	#States /m	#Transitions /m	Time /s
P1.1	-	-	-	OOM
P1.2	True	1.43	2.04	815
P1.3	True	10.8	15.8	7045

Table 5.1: Experiment: Deadlock Freeness Checking

by the using of multi-value global variables. Another observation occurs to us that the system is deployed in the bedroom and shower room where there are no sharing of sensors between the two rooms and the patients' behaviour is exclusive to the rooms. In this case, the complete system state space can split into two sub-system models- *SmartBedroom*() and *SmartShowerRoom*(). We apply deadlock freeness checking separately on this two sub-system models. In the experiment, the modification to the system model is trivial- removing modelling parts related to the other room from the environment model and sensor models. The property are thus specified as the following:

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

As shown in Table C.2[2], both sub-system models are deadlock free. However, the state space of *SmartBedroom*() model is much less than the state space of *SmartShowerRoom*() model as shown in column 3. It is because there are more activities monitored in shower room than bedroom. The more activities to be monitored, the more interleaving will be produced which will significantly boosts the state space.

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.

[2]OOM- Out of Memory

Model	Bedroom		ShowerRoom		BothRooms	
	#Sts /k	Time /s	#Sts /k	Time /s	#Sts /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

Table 5.2: Experiment: Component Deadlock Freeness Checking

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 5.2[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.

5.2.2 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. Rest of the four reminders can be specified in similar ways.

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.

[3]Sts- States

```
#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 *Sitting_Bed_Too_Long* **Reminder (P2.2)**

This property states that when a patient who has troubled sleep by sitting on the bed for too long, the system will always prompt a reminder ask him to sleep eventually.

```
#define TroubleSleep (sensors[PRESSUREBED_1] == SITTING
          && Duration[PRESSUREBED_1] > 30);
#define RemindedSleep (ReminderStage[SIT_BED_TOO_LONG*2 ]≠ 0);
#assert SmartBedroom()  ⊨ □ (TroubleSleep → ◇ RemindedSleep);
```

Here, condition *TroubleSleep* specifies the scenario that someoneis sitting on bed more than 30 time unitess, and *RemindedSleep* defines the state the reminder is prompted.

Guaranteed *Wandering_In_Showerroom* **Reminder (P2.3)**

This property states that some patient is inside the shower room but no activities are detected that the wander flag is set to be true. In this case, the reminder asking the patient to leave shower room will eventually be sent so as to prevent slipping/ falling of the patient.

```
#define WanderingInShowerRoom (Pos_Person[1] == SHOWERROOM && WanderFlag);
```

```
#define WanderReminded (ReminderStage[WANDERING_IN_SHOWERROOM*2] ≠ 0);
#assert SmartShowerRoom() ⊨
          □ (WanderingInShowerRoom → ◇ WanderReminded);
```

where condition *WanderingInShowerRoom* specifies the scenario where a patient is recognised to be wandering in shower room, and *WanderReminded* defines the state the reminder is prompted.

Guaranteed *No_Soap_Used* Reminder (P2.4)

This property states that when the system detects that no soap usage for more than 15 time units while the patient is showering, the reminder contains instructs to use soap will eventually be sent.

```
#define ShowerWithoutSoap (ShowerFlag && Duration[SHAKESOAP] >= 15 &&
          !SoapFlag && Pos_Person[1] == SHOWERROOM);
#define NoSoapReminder (ReminderStage[SHOWER_NO_SOAP * 2] ≠ 0);
#assert SmartShowerRoom() ⊨ □ (ShowerWithoutSoap → ◇ NoSoapReminder);
```

where condition *ShowerWithoutSoap* specifies the situation that the patient is showering which has lasted for 15 time units but there is no soap event sensed, and *NoSoapReminder* defines the state the reminder is prompted.

Guaranteed *Shower_Too_Long* Reminder (P2.5)

This property states that when the system detects that a patient is showering and the shower tap is running for more than 30 time units, the reminder *Shower_Too_Long* asking

patient to finish shower will eventually be sent so as to prevent patient for catching cold or faint.

```
#define ShowerTooLong (ShowerFlag && Duration[SHAKETAP] >= 30
          && Pos_Person[1] == SHOWERROOM);
#define EndShowerReminded (ReminderStage[SHOWER_TOO_LONG * 2] ≠  0);
#assert SmartShowerRoom() ⊨ □ (ShowerTooLong → ◇ EndShowerReminded);
```

where condition *ShowerTooLong* specifies the the condition which decides the patient is showering for too long, and *EndShowerReminded* defines the state the reminder is prompted.

Guaranteed *Tap_Not_Off* Reminder (P2.6)

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.

The results of the verification are shown in Table 5.3. 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.

Property	Result	#States /k	Time /s
P2.1 LyingWrongBed (LWB)	True	808.4	616.8
P2.2 SitBedTooLong (SBTL)	True	798.3	607.2
P2.3 WanderingInSR (WIS)	False	58.24	27.48
P2.4 ShowerNoSoap (SNS)	False	196.6	107.5
P2.5 ShowerTooLong (STL)	False	1018	2635
P2.6 TapNotOff (SNO)	False	701.8	489.1

Table 5.3: Experiment: Guaranteed Reminders Checking

5.2.3 Contradict Knowledge

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;
```

5.2.4 Conflicting/False Reminders

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

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;
```

We tested the system model if it can reach certain false states listed above. They results are shown in Table 5.4. The *False* result in first row says it is invalid that the *Lying_Wrong_Bed* reminder will send to a wrong person. To invalidate a property requires exploring the whole state space. Thus, the time taken in verifying the property *FalseReminders* : *LWB* is much more than verifying the rest of the properties[4]. Furthermore, there are many occasions of the reminders in the system to conflict with each other as shown in row 3 and row 5-10 in the table. This is due to the inability of deciding the correct location of the users. Furthermore, it is obvious that scheduling policy of reminders in AMUPADH needs to be improved.

5.3 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

[4]While verifying reachability properties, the model checker will stop visiting system states as soon as a witness trace is found.

Model	Fault Type	Result	#States /k	Time /s
Bedroom	FalseReminders: 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

Table 5.4: Experiment: Testing Faults

- *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.

Positive feedbacks are given by system designers and engineers. 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 extra PIR sensors in the bedroom and some rules to assure the consistence among context variables.

5.4 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 noticted 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.

In this work, we propose a formal modelling framework for PvC systems. Different modelling patterns are discussed according to the typical features of systems such as concurrent communications, context-awareness and layered architectures. We also provide environment modelling patterns which are usually not considered in modelling 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 modelling 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 tech-

niques) in analysing PvC systems. Possible future works of this research could be applying advanced techniques to tackle the state space explosion problem specially tailored for PvC systems.

Chapter 6

Rule Anomalies Detection: A Model Checking Approach

The challenges facing PvC systems especially for Ambient Assisted Living (AAL) applications are in providing activity recognition of assisted people and also rendering adequate assistive services. It become more challenging when there are multiple users sharing the environment. A detailed interview survey is conducted in [30] showing that in a multi-user environment, activities could be performed in sequential, interleave, concurrent and even conflicting steps. Novel activity recognition approaches are proposed based on multi-model sensor readings such as rule based logic reasoning [82, 29, 74, 7]. For example, if the presence of user, Alice in the shower room is detected by RFID sensor and Passive Infrared sensor while the other sensors remain silent, the system is then aware of Alice's wandering in shower room that she needs help. Their approaches are more accurate and more practical than those relies on images and videos (not accepted in practical for privacy issues).

However, the correctness the rule base remain a non-trivial problem. Incorrect or vague rules could impair the system's capability in determining activities, which could then result

in a lack of, or inappropriate service to be offered. Unreliable rules would also provide a misleading reflection of the actual situation, which is unacceptable in mission-critical or urgent scenarios. The task in correcting relatively large rule repositories is considerably laborious. Therefore there is a need to construct a system that is able to verify and ensure the specificity of rules, and to also automatically correct the erroneous rules. From our experience of deploying a rule based activity recognition system, known as AMUPADH [11] in PeaceHaven nursing home, we identified four challenges:

- Such a rule based AAL system incorporates a reasoning engine that performs activity recognition based on rules that are pre-determined by domain experts, which in this case pertains to nurses and caregivers stationed at PeaceHaven. These rules assist the system in monitoring residents' activities and provide the appropriate reminder prompt to residents through reasoning.

- We observe that although monitored residents perform Activities of Daily Living that is consistent with prior information provided by nurses, there are still occasions where information provided by nurses is insufficient to construct a system that can recognise all the resident's activities. Hence, it is possible that the system could still provide erroneous prompts.

- We determine that these issues are a result of the rules' inability to infer and recognise all activities based only on first order descriptive logic. Rules are also created with logic loopholes due to unawareness of actual scenarios. We are also unable to fully capture factors involved in the changing ambient environment within our rules. Therefore such logic flaws could only be exposed during actual deployments and due to limited granularity and coverage of our sensors, in addition to uncertainties within the ambient environment, the manually defined rules did not perform up to expectations.

- The resulting system thus becomes very difficult to debug because of the relatively

large number of rules and various scenarios that have to be tested in actual deployments to ensure the system is working.

In this work, we propose ACARP, an automatic approach which is based on exhaustive manipulation of all possible scenarios to tackle this rule verification problem. The common rule anomalies are defined by comparing the relations between rules such as redundant rules are rules that have similar conditions. Instead of syntax checking, we discovered that these anomalies can be represented as reachability and liveness properties that can be verified against a system model. Thus, the rules verification problem is transformed into a model checking problem. By adopting the formal modelling framework proposed in Chapter 4, we constructed the scenario model for activity recognition. Further, ACARP automates the rule modelling by translating rules into CSP# modelling language [75] which is an expressive and well supported modelling language. In such as way, the rule anomalies can be automatically detected by reusing the existing model checking algorithms efficiently. Our approach is applicable for general rule based PvC system since the common architecture are adopted.

6.1 ACARP Rule Checking System

Rules and their respective rule engines are considered to be an important part of human behaviour and activity monitoring systems. Rules can be complicated and crucial for real-time and mission critical applications, especially for healthcare purposes. However, it is difficult to manually discover and correct errors within relatively large number of rules. As evidenced in the modelling and verification of the AMUPADH system, rules can be checked using model checking techniques, and counter examples reported by model checkers can be used to identify possible errors in the rules. In the following subsections, we explain how ACARP can parse Drools rules into CSP# and auto-detect rule anomalies.

6.1.1 Drools Rule Engine

The Drools [18] reasoning engine contains rules that can be written using first order descriptive logic. Rules are able to handle scenarios with multiple conditions(IFs) and consequences(THENs). Context-aware reasoning engine [92] is proposed and implemented within the AMUPADH system. However, complicated scenarios with overlapping activities cause logic flaws and some rules are unnecessarily activated. When certain rules are changed, other rules dependent on these modified rules can become redundant. Additional rules that are inserted to complement present rules can also further complicate the system.

We tried overcoming these problems by appropriately classifying the rulesets into different groups of rules. Any modifications made to a group of rules would affect only rules within the particular group. However, this is not a perfect solution as there would still be situations where rule conflicts would occur between different rule groups which are used to monitor the entire smart home. Therefore the model checker PAT would be used to help resolve this problem by performing rule checking and verification.

6.1.2 System Workflow

As shown in Figure 6.1, the system is designed to incorporate a parser that takes Drools rules and Java classes as input, so that the system can automatically produce a model in PAT. PAT subsequently checks the newly-created rule model against the properties defined in Section C.6. If some properties are satisfied, a counterexample will be reported. PAT then analyses and traces the report to locate the erroneous rules and autocorrect them according to predefined logic, or the system can also alert users to perform manual correction.

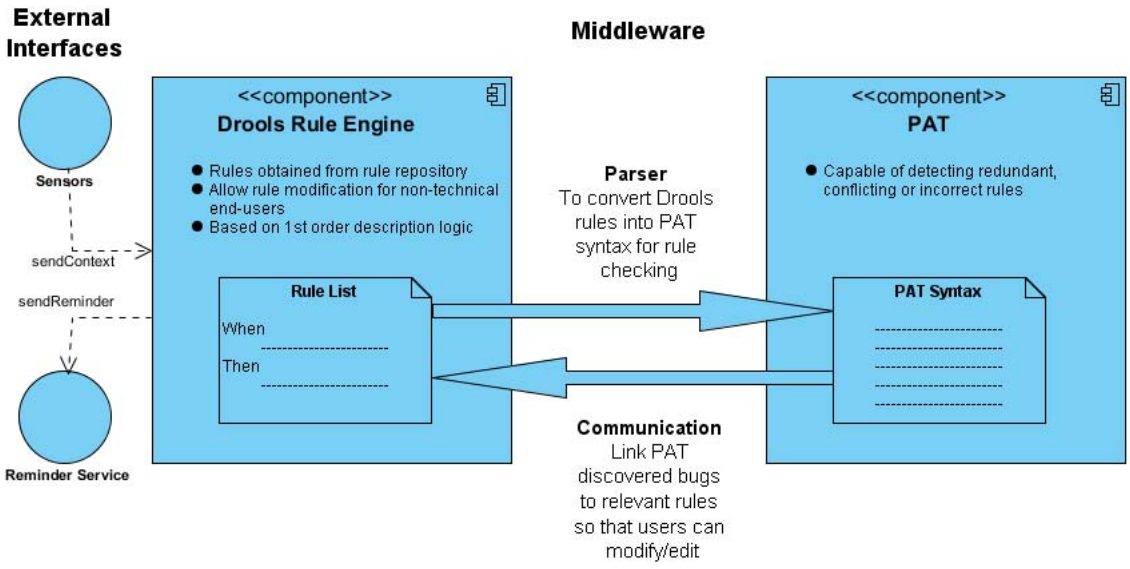


Figure 6.1: Auto Correction System Component Diagram

6.1.3 Translating Drools rules to CSP#

We use Drools because it is a popular rules platform widely adopted in research fields and industries. Drools Expert[1] is an open source rule engine maintained by JBOSS community. It is broadly applied in recommendation, financial security and other business systems. The language syntax of specifying Drools rules is based on first order descriptive logic and external methods and classes written in Java can easily be used in Drools. PAT supports both of these features such that first order logic can be directly mapped to its form in CSP#, and Java classes can be parsed into C# external library, which can be called in CSP# models. Therefore, we parse Drools rule syntax into CSP# in PAT.

For usage of temporal rules with our approach, users are required to use the Real-Time System (RTS) module within PAT. Our approach assumes sensor inputs are true and without uncertainty. The full syntax of Drools is quite expressive and the general pattern is shown

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

in the example below. Interested readers are encouraged to read [18] and [92] for details.

```
rule "Person lies on BedA, Wrong Bed"
    when   s1: Sensor(id == "RFID", name != "PersonA")
           Sensor(id == "BedA", state == "LYING")
           msg: MessageInterface()
    then   msg.send("Error: " + s1.getName() + " is lying on wrong bed.");
end
```

The above rule example obtained from the AMUPADH system clearly defines the conditions and consequences describing the situation of a person lying on the wrong bed. The bed has a RFID reader and pressure sensors installed. When a person other than "PersonA" uses the bed and registers a pressure status of *LYING*, an alert message is sent . Note that the symbol "s1" gives a name to an object or field that can be later quoted in the scope of this rule and "Sensor" can be viewed as a field or a point for external method call or object reference.

Parsing DROOLS rule to CSP#

After identifying the general pattern of Drools rules, the mapping of rules are defined in this section.The rule engine consists of the main rule file and the Java classes that specify global variables and external methods. We manage the parsing process as listed below:

Step 1: Extract Shared Information For the purpose of easy management, the shared information is declared and kept in Java classes instead of rule files. This information includes the global variables and constants.

Step 2: Mapping Rules into CSP# The parser processes the rules one at a time and splits the rule into three parts, namely the 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, and the consequences into *ifa* statements, so as to point to the next rule.

The two-step parsing tool automates the process of modelling rules and reduces the time required for verifying rulesets. Human intervention might be required in rewriting certain Java classes into C#. However, experience shows that this effort is minimal, since Java classes are seldom changed during system development.

Users who understand the context and usage of the rules are most suited for conducting this process as the generated model's performance is dependent on the user's expertise in accurately translating rules and inclusion of additional embedded semantics from Java classes into the PAT model. However, our approach does not directly support the modelling of rule priorities, but there are extensible work in progress. The choice of using a different rule engine is also possible, but it would need to adhere to PAT's input requirements.

6.1.4 Detection of Rules Anomalies

We model anomalies as properties within PAT. Based on results and feedback obtained from the AMUPADH system deployed at PeaceHaven nursing home, we identify the following critical properties:

– *Property 1: Non-reachable rules*

Non-reachable rules are trivial as some rules' conditions are never satisfied during ruleset execution. 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. *Detection* of these rules is done by checking if

every rule is reachable during the exhaustive search of system state space. *Correction* by automatically removing such unnecessary rules is offered by the ACARP system.

– *Property 2: Redundant rules*

Redundant rules are occurrences of multiple firing rules at the same system states which producing identical or different system result. We define two kinds of redundant rules, *duplicated rules* and *subsume rules*. The former refer to rules whose condition parts are the same, whereas the latter applies to rules where the conditions of one rule is a subset of the other. The *detection* of these rules can be done by checking if the two rules in question are always fire together. We perform *correction* on these rules, by removing one of the rules if both sets of conditions and consequences are identical. If the consequences are different and not conflicting, we merge them into a single rule and classify the remaining rules as conflicting rules.

– *Property 3: Logically Conflicting Rules*

Logically conflicting rules are rules that satisfy all conditions at a particular state, but the consequences are not logically sound and thus are conflicting. This property cannot be easily detected in routine system tests and the possibility in finding this property using additional model checking techniques is still dependent on the designers' experience in finding such rules. We *detect* this error by checking if possible pairs of conflicting rules can happen at the same time and analyse the rule traces, so as to point out the conflicting pair of rules for manual *correction*.

6.2 Experiments and Discussion

We tested our implementation on two rulesets, which are rules used for two respective areas, namely the bedroom and shower room. As shown in Table 6.1, the two rulesets contain rules

Scenario	#Rules	#Non-reachable	#Redundant/ #Duplicated	#Conflicted
Bedroom	17	2	8/3	2
Shower Room	22	5	16/2	-
Avg. Time /s	-	2.05	3.05	-

Table 6.1: Experiment: Anomaly Detection in Activity Recognition Rules

that are paired and tested for anomalies of non-reachability, redundancy and logic conflicts. The advantage of using a model checker like PAT is that it provides feedback consisting of counterexamples or traces of actions if any anomaly is detected, which is very useful for analysis of our system. The traces can point out names of erroneous rules and their respective violated properties. The activity sequence that leads to the property violation will also be listed out.

From the experiments, we found cases of the following anomalies. In the bedroom scenario, our system detected a rule that was non-reachable even after an exhaustive simulation of all possible situations. We realised that it was a rule defined to recognise an activity of opening a cupboard. However, in that particular state of the system at the time of detection, there was no sensor deployed to detect such an activity. We subsequently confirmed that rule developers forgot to remove this rule from the repository even after the use case of recognising cupboard usage has been previously abandoned. During the test for redundancy, ACARP discovered five duplicated rules which were accidentally added into the rule repository for testing and were not removed due to negligence.

Multiple pairs of logically conflicting rules were discovered within the rulesets used for our PeaceHaven deployment of the AMUPADH system. We also observed multiple reminders that were simultaneously prompted to the same user and we tried replicating such conflicting scenarios using the model checker PAT. The verification result with regards to the reachability of this defined scenario turns out to be valid within our model, thus providing

us with a trace that denotes 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. We were able to trace this situation using our ACARP system.

Discussion From the experiments, we discovered the stated anomalies as described by our defined properties. They could be caused by changes made to the scenarios or rules. It is impractical to manually examine and verify the validity of relatively large rulesets. Moreover, some existing rule verification techniques can only perform simple syntax checking which fail to detect logical conflicts. The other techniques and methodologies also could not be used in our system due to the high level language features used within modern rule engines, in addition to the lack of general tools and support. Hence, our approach is required in order to provide features for modelling and also to effectively detect the defined anomalies and be able to perform customised verification for other testing purposes.

6.3 Related Work

In rules verification, anomalies are identified and classified into different categories. Various algorithms are proposed to detect anomalies in order to build robust systems. Ligeza and Nalepa [43] reported state-of-the-art rule representation and types of inferences. Taxonomies of rule anomalies are proposed regarding redundancy, consistency, completeness and determinism. However, their rules are simple compared to context-aware rules that are used in our system. In addition, there was no discussion regarding the detection and resolution of anomalies too.

Drools verifier [1] from JBOSS community also performs rules verification. This tool searches for anomalies such as redundancy and subsumption but their approach is based

on merely syntax analysing and it lacks the ability in discovering high level semantic errors such as conflicting rules.

Preece et al. [62] surveyed the verification of rule based systems focusing on detecting anomalies. They described four types of anomalies regarding redundancy, ambivalence, circularity and deficiency. Five rule verification tools are compared based on their capability of detecting the four anomalies. They provide insights on underlying principles of rule verification and state of the art in building a tool for carrying the task. However, their definitions for anomalies and the surveyed algorithms are not directly applicable to our system. Firstly, their methods are based on the syntax and semantic logics between rules instead of the rule effects on the system behavior. Secondly, the algorithms are mostly designed for goal driven rules, such that the conditions and consequents of a rule have strong causal connections and consequences are true only when all conditions of a rule are satisfied. This might not necessarily be true in our ACARP system.

Researchers in [19] used Ambient Calculus to model a location sensitive smart guiding system in a hospital. The mobility issue is well modelled and reasoned in their work using modelling methods, but without adopting a rule-based approach. Our work is able to adopt hierarchical modelling languages which is also supported by model checkers like PAT for automatic verification.

Therefore, new methods for analysing rules in our ACARP system are needed in consideration of our context-aware rules.

In this work, we presented a system that is able to use model checking techniques and the model checker PAT in performing rule verification of activity recognition rules within a smart home environment. We used Drools engine and tested our rules within a deployment scenario at a local nursing home. In order to ensure validity of the rules, we created a model of the rules and checked for non-reachability, redundancy and logically conflicting

properties of the rules. These three important properties allow us to evaluate the rules and also assist in automatic rule correction, therefore reducing reliance on human intervention and the time needed for verifying the entire ruleset.

Chapter 7

A MDP-based Approach for Reliability Analysis

One well-known application of PvC system is to assist elderly/ disabled people to live independently at their homes such as AMUPADH system. In the system, various hardware devices and software components are cooperating together to produce a correct reminder service. For examples, multiple sensors and an inference engine is used to perceive environment changes and user intentions. Small home network is setup for communications between distributed components. Actuators such as bluetooth speaker, IP TV or iPad are used for prompting interactive reminder services. In general, a PvC system is considered to be reliable when all required services are successfully delivered at the right time to the right person. However, there are many causes leading to system failure, e.g. a critical sensor fails or a network node malfunctions causing important messages being lost. Due to limitation of current technology, such failures are unavoidable, thus it becomes critical to analyse and manage the reliability in an acceptable rate.

Analysing the reliability of a PvC system is not trivial. As it is well-known, PvC systems

are inherently complex. They are composed of multi-layers of software and hardware components which have limited capability and accuracy. Researchers in [59], [65] reported that the often inherent inaccurate and unreliable low-level sensors are used to detect context information from the environment. This is probably due to cost efficiency considerations, i.e., less sensors or low-capability but cheaper sensors are selected due to budget constraints. Moreover, PvC systems are often rule based where rules are manually constructed by engineers with limited expert knowledge. Sometimes, two different scenarios are characterised as similar cases because the rules defined to recognise the scenario is similar. For example, a rule defined to recognise the abnormal behaviour, showering-for-too-long, may have similar condition with the rule defined for tap-not-off . Both of the rules depend on the same sensor detecting shower tap on for more than certain time. It often leads to wrong and confusing results. Wireless communication is unstable in the system as well. For instances, Zigbee which is a low-cost, low power, wireless mesh network is adopted for communication among sensor and network nodes such as bridge or router go down from time to time. Human errors like a user forgets to wear the sensible tags could also causing the failure of the system. Besides the complexity of the system, the feature of non-deterministic behaviour hinders the application of most existing approaches. In PvC systems, information detected from multiple sensors are combined in order to recognise a user activity. For example, to detect the showering without using soap behaviour, the shake sensors on tap and soap dispenser are needed. However, due to the unpredictable user activities, it is impossible to decide a particular sequence of sensor activation. For instance, after entering the bedroom, the user might sit on the bed, triggering the bed pressure sensor or he might go to washroom directly triggering the RFID sensor for identity tracking. Thus, the order of sensor triggering is non-deterministic. All in all, an approach which is able to handle both the complex architecture and nondeterministic behaviour is needed.

Probabilistic model checking gain great importance, especially for such complex system with

non-deterministic behaviours. Reliability analysis is an useful application of this technique. Some properties are of interest to end-users and developers of PvC system like "what is the overall system reliability?", "to reach a certain overall reliability, how reliable should certain sensors be?" and so on. Reliability analysis by modelling system architecture with Markov Chains is first proposed by Cheung [14] in 1980. It has been applied in various case studies, e.g., Gokhale et al. [26] analysed the SHARPE tool for stochastic modelling by constructing a DTMC and found out the relation between system reliability and fault density per component (subsystem). Goseva et al. [27] performed a case study on a system of the European Space Agency including reliability analysis and sensitivity analysis. Wang et al. [85] analysed stock market system by construct DTMC and predicted the system reliability. However, all these case studies are software systems. To the best of our knowledge, there is no reliability analysis has been conducted on any PvC systems which involves not only software systems but also networks, sensors and human activities. In such a complex system, probability distribution of transitions among system components are not able to obtained in most of the cases. MDPs allowing use of nondeterministic choices to model hard-to-predict transitions are adopted for modelling of PvC systems.

In this work, three general but highly important questions related to the reliability of PvC systems are investigated. First, "what is the overall system reliability if reliability of all its components and subsystems are known, considering all possible user behaviours, and unreliable factors?". This is referred to the problem of *reliability prediction*. This question is to be answered necessarily before system deployment since end users would prefer to know how reliable the system is. Second, "what is the reliability required on subsystem or some devices if there is an expected reliability on overall system?". This is referred to the problem of *reliability distribution*. Addressing this issue is pretty useful because we can have specific quantitative requirement on designing software subsystems or selecting hardwares, whose quality are highly related to cost in most of the time. Last but not the least, we are

interested in finding out the most critical parts to system reliability, based on quantitative measurement, that relatively more efforts and fund can be spent on? This can be done based on *sensitivity analysis*.

7.1 System Modelling using MDPs

Compared to DTMC, MDP allows us to capture both probabilistic and nondeterministic behaviors. A central issue is: when to use nondeterministic choices and when to use probabilistic choices. In general, *probabilistic choices can be viewed as informed nondeterministic choices.* That is, we use a nondeterministic choice when we have no definitive information on how the choice is resolved. For instance, if all we know is that there are two different outgoing transitions after executing a component C, we model the two transitions using a nondeterministic choice. If the choice is made locally and we are aware of each outgoing transition, we can model C with a probabilistic choice. However, if the result of executing C is correlated to its inputs, there are two cases. If the inputs are the result of executing some other component K in the system, we may either model it as a nondeterministic choice conservatively; or we calculate the probability distribution of C's results based on the probability distribution of K's results. If, however, the inputs of C are from an external environment which is difficult to predict (e.g., like the traffic of stock transactions), a nondeterministic choice would give us a "safer" model.

There are three major elements in an MDP model for PvC systems, i.e., the nodes, the transitions and the reliability values. In the following, we take the model in Figure 7.1 as an example to explain the modelling technique.

Nodes

Typically, in a general PvC systems, the sources of unreliability could be failure of sensors and network devices, error in softwares and connection loss/transmission failure in networks. Thus, in an MDP model of a PvC System, nodes are abstractions of sensors, software components and network devices. To decide which device/ component are necessary to be modelled, we need to analyse the rules which is used for reasoning. For example, in TNO case, four sensors are related for recognising this behaviour. Besides, there are multiple choices of playing this reminder e.g., playing on iPad, on Wi-Fi network or on Smart Phone through 3G network. Thus, the four sensors, iPad, smart phone, Wi-Fi network and 3G network need to be included in the model as nodes. Similarly, the Zigbee network, mini server and rule engine are related.

In Figure 7.1, there are three types of nodes, i.e., circle nodes denote for sensors, square nodes for hardware devices including software component residing inside, and cloud shape nodes for networks. Double circled nodes are accepting nodes where all of them are success nodes which are abstractions of displaying devices of reminders expect for the failure node. The different shapes of nodes are for demonstration purpose, in an MDP model, they are treated the same.

Transitions

In PvC systems, there are usually two types of relations between nodes happening before and message forwarding relations. By analysing the reasoning rules, we are able to extracting the relations between sensors. Happen before relation usually exists among sensors. It is decidable when there is a particular order of triggering sensors. For example, in Figure 3.2, the RFID reader is placed near the door. It is triggered earlier than other sensors in the room if the system starts with all users outside. Thus, in the MDP model, it should be

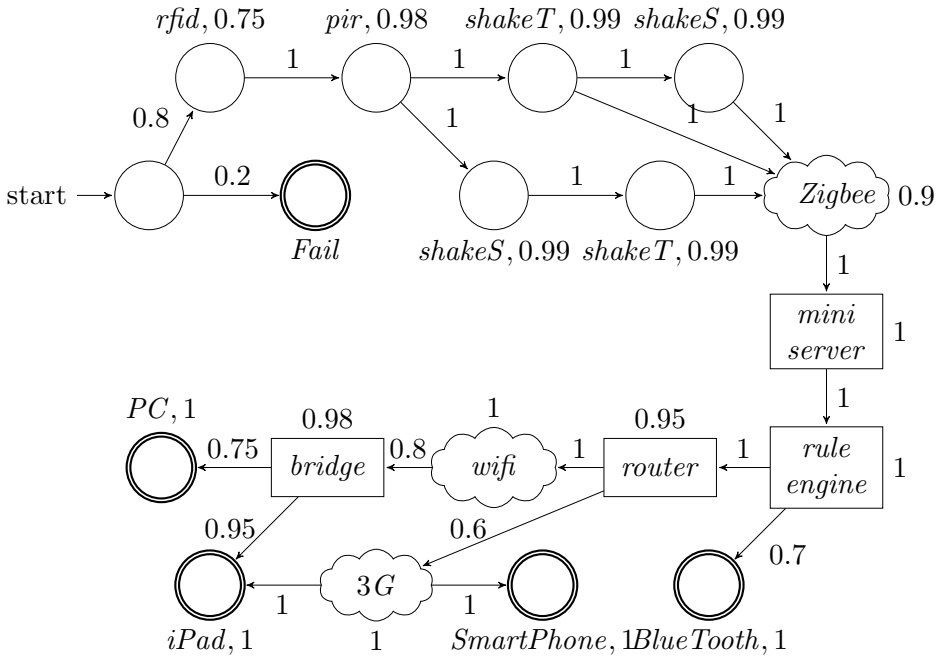


Figure 7.1: Bathroom Scenario- TNO: Tap Not Off

placed in front of the rest of sensor nodes.

However, sometimes, the happen before relation is not deterministic. For instance, in the model of Figure 7.1, there is no specific orders between shake sensors on the tap and soap dispenser. Thus, we need to enumerate all the possible orderings. Besides, there is one rule decides the behaviour on shake sensor on Tap solely among the two shake sensors. Thus, there is a transition link from *ShakeT* to *Zigbee* making the ordering asymmetric. We suggest that it is better to enumerate all the possible transition orders in the initial model, especially when there are multiple rules defined based on similar sensors.

As for message forwarding relations, they are decided in the system design. For example, in the TNO model, the messages are sent to the mini server via Zigbee network. Thus, Zigbee node is placed between the sensors and mini server with transitions indicating the sequence of messages transmission. Similar rules apply for rest of the transitions.

Reliability Labelling

One final step is to label the nodes and transitions with reliability values. It is usually provided by system engineers. Nodes are labelled with reliability values of the corresponding devices. For transitions, there are different cases. At the initial node, the outgoing transitions usually representing the user behaviours. In the TNO case, there is 20% of time, the user will throw the sensible tags away (result drew from an experiment conducted by the engineers). Thus, initially, there are 0.2 probability leading to system failure directly as the user identity is unknown. Additionally, the happen before relations are usually non-deterministic choices with no specific probabilities, thus by default, we assign the value 1. As for forwarding relations, due the the signal strength, transitions to/ from network nodes have different reliabilities. For example, in Figure 7.1, transitions from Wi-Fi node to bridge node has the reliability of 0.8 since the bridge is placed on the wall outside the bedroom. The nurse PC in common area is further away from the bedroom, thus the transition from bridge to PC is as low as 0.75.

Fortunately, AMUPADH system has been deployed in a real user environment for data collecting, the engineers are able to provide realistic estimations of device reliabilities based on event log analysis. Since the reliability data varian on different days, we choose the best effort ones.

7.2 Reliability Analysis Problems

Based on the MDP model constructed in above section, three interesting directions of reliability analysis is demonstrated. Figure 7.2 shows the workflow of this approach. The approaches for reliability prediction, distribution and sensitivity analysis are shown in Figure 7.2 (a), (b), (c) respectively. In the following, each of the approaches is demonstrated.

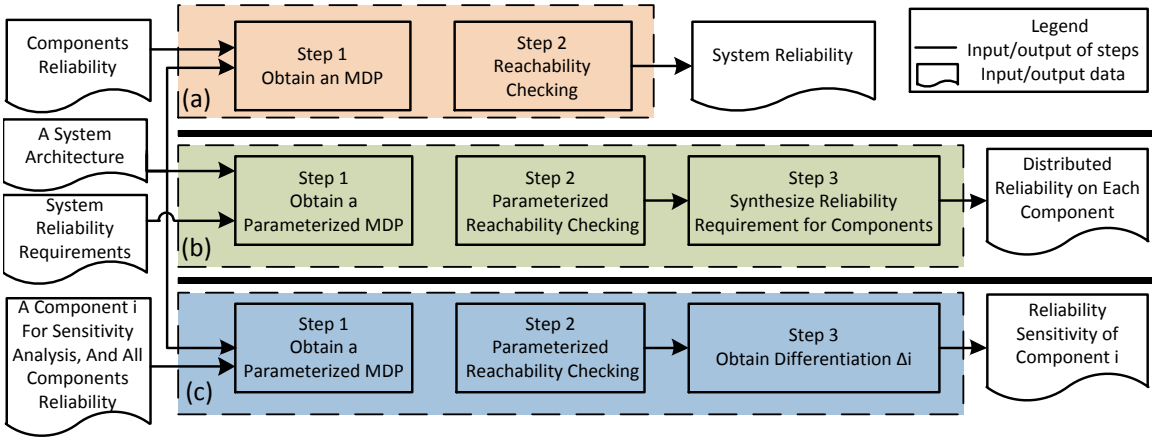


Figure 7.2: Workflow: (a) reliability prediction; (b) reliability distribution; (c) sensitivity analysis

7.2.1 Reliability Prediction

As shown in Figure 7.2 (a), the reliability value of each component and an MDP model of a system are required for calculating the overall system reliability. This is equivalent to check the probability of the system never fails. Different methods have been developed to calculate the maximum and minimum reachability probabilities, i.e., the probability of reaching to the accepting states, denoted by $Pr(M, a)$. Value iteration as introduced in Chapter 2.2.3, is adopted for its better performance than most linear programs. Predicting the reliability of the overall system based reliabilities of components is essentially calculating the probability of reaching a successful state from the initial state.

There could be multiple paths in a MDPs model where each path is considered as a scheduler. The reason for this is that there are multiple success state in addition to nondeterministic choices of outgoing transitions. As a result, the measurement of interest is thus the maximum and minimum reachability probabilities. Let b_i be a state in the set of success state B. The lower bound of the overall system reliability is the minimum value of $P^{min}(\mathcal{M} \models \diamond b_i)$ for each b_i in B. Similarly, the upper bound is the maximum value of $P^{max}(\mathcal{M} \models \diamond b_i)$ for

each b_i in B.

7.2.2 Reliability Distribution

The approach on distributing the overall system reliability requires two inputs: (1) a reliability requirement R on the overall system; (2) a parameterised system model in the form of an MDP with weights for the components participating in reliability distribution. The workflow is shown in Figure 7.2 (b). The goal is to find a reliability requirement on each component whose reliability is unknown so that the overall reliability requirement is satisfied. The resultant requirement on component c is in the form of a reliability probability R_c.

Given an MDP M and a scheduler δ, we can obtain a DTMC M_δ. Assume the assigned weight for component (in reliability distribution) i is w_i, which is also given. Therefore, each component reliability requirement can be expressed as $w_i x$, where x is a variable. Therefore, the system reliability (i.e., $Pr(M, a)$) is a polynomial function constituted by variable x only. Using numerical methods like Newton's method, we can obtain a lower bound on x, which is the reliability requirement we need. For multiple schedulers and we need to guarantee that the system reliability requirement is satisfied with any scheduler. We compute a lower bound on x for every scheduler and the maximum of the lower bounds gives us the minimum requirement on component reliability. Based on [9], only finitely many memoryless schedulers need to be considered. Our algorithm works as follows. First, an unvisited memoryless scheduler δ is selected. Next, using δ, we perform the value iteration method on M_δ. The following shows how the result vector V is updated. Assume scheduler δ chooses a distribution μ_s at state s: $V^{(n+1)}(s) = \sum_{t \in S} x \times \mu_s(t) \times V^{(n)}(t)$. Once a stopping condition is satisfied, we obtain a constraint $V(init) \geq R$ and solve the equation $V(init) - R = 0$ using Newton's method to obtain a lower bound on x so that $V(init) \geq R$ is true. The steps above are repeated for all memoryless schedulers.

The number of total memoryless schedulers equals to the product of the numbers of distributions for each state, e.g., if there are ten states and two of them both have 3 distributions and the rest has one, the number of schedulers is 9. Essentially, the more nondeterminism there is, the more schedulers are to be considered. In practice the number of schedulers are manageable as we are dealing with a high-level system model. Further, since each scheduler is considered to be independent, we can parallels the computation.

7.2.3 Sensitivity Analysis

Sensitivity analysis indicates how to improve the overall system reliability efficiently and effectively under a circumstance of limited resource. For example, if a system is shown to be not reliable enough, based on each components reliability and the system architecture, how to prioritize the components such that reliability improvement of a higher priority component would result in more system reliability improvement.

The workflow for sensitivity analysis is shown in Figure 7.2 (c). In general, the sensitivity s_i of the system reliability R with respect to the reliability R_i of i^{th} component is defined as the partial derivation of system reliability, denoted by f with respect to R_i, as in the form $\Delta_i = \frac{\delta f(R_1,R_2,...R_i,...R_n)}{\delta R_i}$. However, analytical solution is hard when system is large and non-deterministic. In our work, we only consider sensitivity analysis on one component each time given all other components' reliability. The sensitivity analysis problem is reduced to calculate $\Delta_i = \frac{\delta V(init)}{\delta R_i}$. $V(init)$ is obtained as the same way as the one in reliability distribution but there is only one variable in this case.

7.3 Analysing Reliability on AMUPADH system

In the beginning of AMUPADH project, we spent three months paying visits to the nursing home for collecting requirements. By observing the patients daily life and multiple inter-

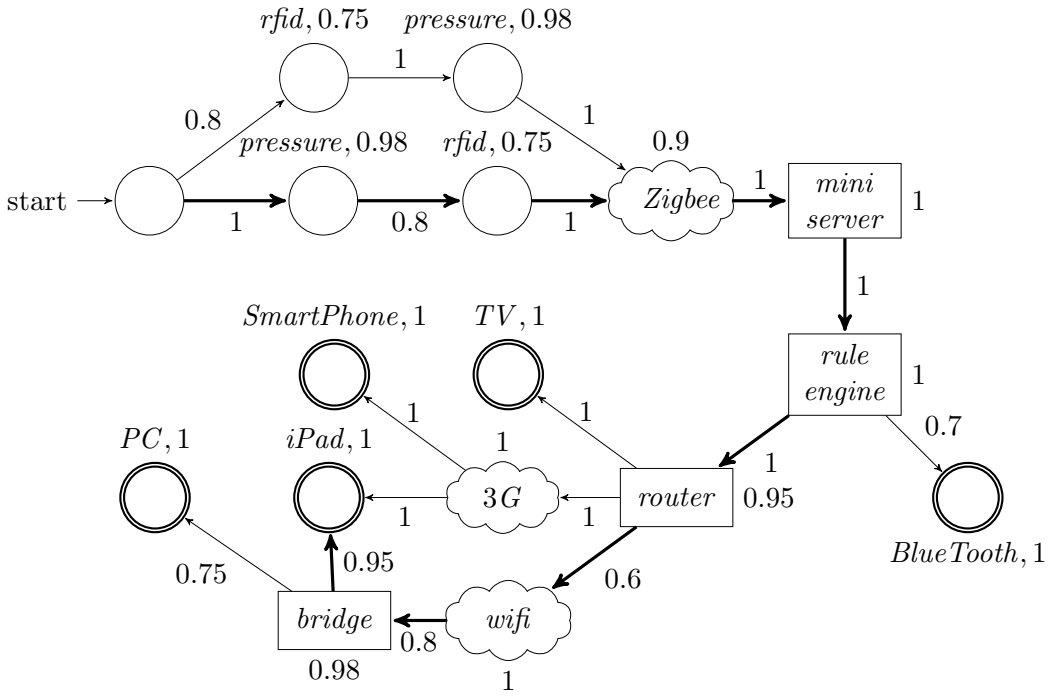


Figure 7.3: Bedroom Scenario- UWB: Using Wrong Bed

views of nurses/doctors, two critical issues with dementia patients (caused by constantly short memory loss and initiation problem) are raised that are sleeping behaviour in bedroom and showering behaviour in bathroom. Furthermore, as introduced in Chapter 3.3, six typical abnormal scenarios that requires reminder help or caregiver intervention is revisited in the following:

- **UWB: Using Wrong Bed** Since a room in the RLA is shared by 2-3 people, the elder patient, especially new residence, tends to lie on a bed without recognising whether its his/her own bed.

- **SBTL: Sitting on Bed for Too Long** Some of the agitated patients often have sleeping problems. They are easily bothered and irritated by what is happening in the environment. A typical symptom is that the patient will get up at midnight and

sit on the bed for very long time until assisted by nurses/caregivers.

- **SNS: Shower No Soap** Due to memory loss, dementia patients constantly forget the normal steps of performing daily activity. In the taking shower activity, the patient could forget what to do next right after the shower tap is turned on. It is reported by the nurses that some of the patient finish the shower very fast without applying soap. Concerned about the personal hygiene, patients presenting this behaviour need to be helped.
- **STL: Showering for Too Long** Similar to the SNS issue, some patients will standing under the shower head for a long time. It is a critical issue that exposing in the water for a long time could cause the patient black out. If not helped immediately, it will even causing death to the patient.
- **TNO: Tap Not Off** It is often the case that dementia patients forget to turn off the tap after showering. In order to save water and energy, this scenario is also detected and reminded in the system.
- **WiW: Wandering in Washroom** Caused by initiation problem, it is possible for the patient to forget at any step of the taking shower activity. Thus, a wandering behaviour is also typical and patients need to be assisted in such cases.

In fact, taking shower turns out to be the most concerned issue of nursing elderly dementia patients. In PeaceHeaven, the nurses need to monitor the showering activity of every patient. Considering the ratio of nurses to patients is 1:15, this shower monitoring creates heavy burden to nurses. After a careful analysis, both the SNS and STL scenarios could be monitored by the system. A two-level reminding solution is thus provided in AMUPADH that when the system recognises an abnormal behaviour, it will prompt a reminder to the patient. If the behaviour remains, an alert will be sent to the nurse's mobile phone to raise her attention.

Rel.	UWB	SBTL	SNS	STL	TNO	WiW
Schedulers	32	24	32	16	64	16
Max	0.3744	0.4190	0.3670	0.3707	0.3707	0.3707
Min	0.2956	0.2463	0.2897	0.2927	0.2897	0.2927
Time	<1 ms					

Table 7.1: Experiment: Reliability Prediction

7.3.1 System Modelling using MDP

In practice, it turns out to be unrealistic to model all the scenarios in one MDP model considering the complexity and readability. Since scenarios are independent with each other, we construct the MDP model separately for each scenario. Based on the modelling approach introduced in Section 7.1, we present the MDP model of UWB scenario in Figure 7.3 as an example. In this case, the system heavily relies on the RFID reader to detect the identity of the user who is using the bed. As shown in Figure 3.2, the RFID reader is placed at one side of the bed. Since it is undecidable that which sides the user will choose to get on bed, it is necessary to enumerate all possible orderings of RFID reader node and pressure sensor node. As there is always a 20% chance of failure associated with RFID reader due to the user throwing away the sensible tag, thus the model is asymmetric for different ordering of the nodes.

7.3.2 Reliability Analysis Experiments

We conducte the experiments on six scenario models on a normal PC with 8 GB memory. All the experiments are finished within seconds.

Reliability Prediction

As shown in Table 7.1, the reliability of six scenarios ranges from 25% to 40% with different scheduling[1] which is quite low considering using the system at home with no human supervision.

One general observation from this experiment is that the system highly relies on the RFID sensors for identity tracking. However, the RFID sensors have the lowest reliability among all the sensors. In fact, due to budget issues, the RFID reader used in the system has a half meter detecting radius which makes it cheaper than others with a larger radius. Besides, these sensors usually involve human errors such as disposing the sensible tag. Especially for elderly dementia people since they are not very stable, the system cannot expect to rely on them to provide critical information. Thus, our experiment result suggests the engineer to replace this sensor for one with a larger detecting range or one does not require a tag.

Reliability Distribution

Further, we explore the reliability distribution on some nodes upon a overall reliability requirement. Two groups of nodes are tested which are sensor nodes and network related nodes. By fixing reliability of the network related nodes, we calculated the distribution on sensor nodes and vice versa. We consider a uniform distribution (where all the nodes have the equal weight) among sensors since they have a relatively similar reliability.

As shown in Table 7.2, it requires each network related node to have a reliability of 0.913 in order for all the scenarios to achieve a reliability of 0.4. However, it is impossible when the requirement raises to 0.5. By a careful examination, we discover that the rule defined

[1] A scheduling in the model denotes for one possible path from the starting node to one success node. Since there are non-deterministic choices in the model, there are multiple possible scheduling.

Req.	Nodes	UWB	SBTL	SNS	STL	TNO	WiW
0.4	Network	0.854	0.904	0.913	0.911	0.911	0.911
	Sensor	0.886	0.938	0.941	0.923	0.923	0.923
0.5	Network	0.914	-	0.965	0.963	0.963	0.963
	Sensor	0.996	-	0.995	0.994	0.994	0.994
Time /s		3.45	2.68	3.86	1.87	11.00	2.35

Table 7.2: Experiment: Reliability Distribution

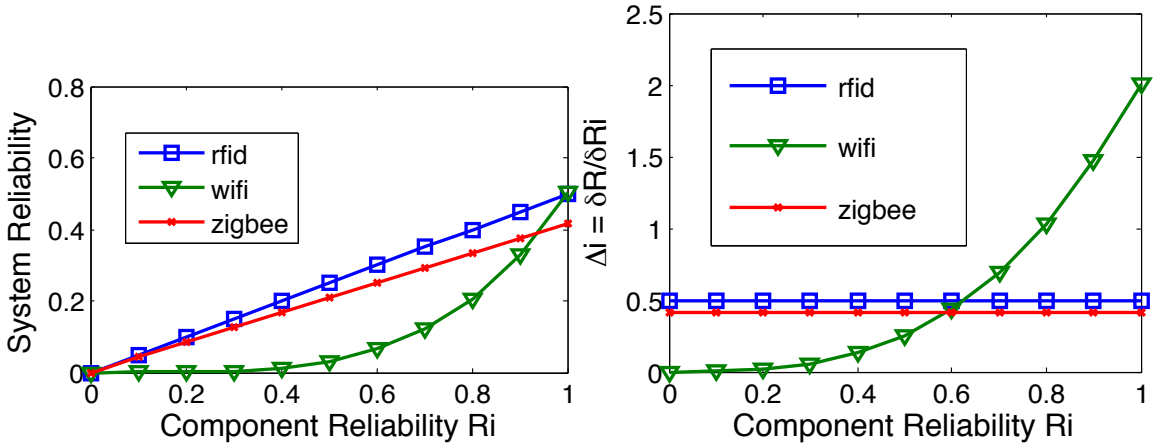
for SBTL has an error. Because the engineer failed to put the user's identity information into the rule's condition, this reminder will have a half chance be sent to the wrong user.

At this point, it is intuitive to ask the question that which node or group of nodes affects the system reliability heavier than the others? If improvements are made on such node(s), it will be more efficient. Thus, we seek the answer from sensitivity analysis.

Sensitivity Analysis

There are multiple schedulers in each MDP model as shown in Table 7.2. We choose a typical one for demonstration and omit the rest due to page limit. As shown in Figure 7.3, the model connected in thick black links are the target schedule. It includes the most features of other schedulers e.g., rely on multiple sensors and the RFID reader with lowest reliability. The iPad case is chosen since playing reminders on iPad is the most common way in practice.

Two nodes and a bundle of nodes are chosen for the experiment which are RFID reader node, Zigbee network node, and bundle of nodes related to Wi-Fi network (their reliabilities are dependent). Figure 7.4a shows the reliability distribution on these nodes. As we can see, improvement on RFID reader node and Wi-Fi bundle can achieve a higher reliability than Zigbee node. Figure 7.4b further suggests that when the reliability of these nodes are greater then 0.7, increasing reliability of nodes in Wi-Fi bundle can achieve better



(a) Distribution (b) Sensitivity

Figure 7.4: Experiment: UWB- Sensitivity Analysis on Nodes

improvement than other nodes. In practice, increasing the reliability of a network might be cheaper than purchasing a sensor with higher reliability, e.g., placing more bridges along the path.

To summarise, these experiments are able to give a good estimation on the overall system reliability. Additionally, it provides useful guidance on improving the system efficiently, especially in budget concerned systems. However, our approach requires knowledge on modelling system in MDP models which make it difficult to use for engineers without necessary background.

7.4 Related Work

Since Cheung's work in [14], several models and approaches on software architecture based reliability evaluation have been studied and compared. Comprehensive surveys on the existing approaches can be found in [36, 28, 25]. Compared to the above work, our approach handles systems with model parameters which are hard to obtain.

Reliability Allocation vs. Reliability Distribution Reliability distribution problem investigated in this work is similar but slightly different from the reliability allocation problem studied in [55, 60, 35]. Given a Markov Chain model of the system architecture, reliability allocation focuses on testing resource allocation to achieve optimal performance. A well known reliability allocation deals with the setting of reliability goals for individual subsystems such that a specified reliability goal is met [40]. While ensuring that a system is sufficiently reliable, the allocation is to find an appropriate set of reliability measures apportioned to each component, based on some optimisation goals, such as minimising the amount of testing time [60, 48]. [48] discusses the reliability allocation which is to minimise the number of remaining faults given a fixed amount of testing efforts. Different from algorithms on allocating testing resource [60, 35, 48], our method on reliability distribution focuses on minimising component reliability requirement, while fulfilling system level reliability requirement.

Reliability Analysis of Conventional Software Systems vs. Pervasive Computing Systems [66] extended the scenario specification (based on Message Sequence Charts) to produce the reliability model the component-based software systems and used Cheung's model for reliability calculation. However, PvC systems are very different from the software systems studied in their work. In fact, non-deterministic behaviours commonly exists in the system that Cheung's approach had no direct supporting modelling formalism. Furthermore, due to those non-deterministic choices, it is not often the case, probability of all outgoing transitions from a source state sum to one as required in Cheung's model.

This chapter demonstrates our reliability analysis work of PvC systems using MDP based modelling and verification approach. The models are manually constructed from the design and implementation of the systems. Three groups of experiments are conducted to answer the questions of "What is the overall system reliability with known reliability value of each

nodes?", "To reach a certain overall system reliability, how reliable should the sensors/networks be?" and "Which node (could be a sensor or network device) affects the overall reliability the heaviest?". Experiments show surprising results that the overall system reliability is below 50%. It is also suggested that to improve the reliability of Wi-Fi network will be more efficient to improve the system reliability. In future, approaches for acquiring reliability of system components are of interest such as hypothesis testing.

Chapter 8

Conclusions

8.1 Summary

In this thesis, we have investigated ways of applying model checking techniques to analyse complex PvC systems. In particular, we have contributed in the formal modelling of the system and verification of its correctness, reliability and real-time requirements.

First of all, a formal modelling framework is proposed by studying the common architecture of PvC systems. General modelling patterns are proposed using formal modelling structures. The framework includes the modelling for both the environment including the human behaviours and system design such as concurrent communications, context reasoning behaviours etc.. Furthermore, by investigation of stakeholders (Doctors, Nurses, System Engineers etc.), critical system requirements are identified. They are transformed from natural languages to formal properties such as safety and liveness properties which are checkable using existing model checking algorithms. This approach is applied to AMUPADH system which adopts CSP# modelling language and PAT model checker for verification. Several unexpected bugs are revealed and we are able to locate the sources of these bugs with the

help of counterexamples provided by the model checker.

Secondly, we demonstrate the work of automatic detection of rule anomalies. A tool is developed to translate the popular industrialised rule language Drools Rules to the modelling language of CSP# by providing the mapping relations of language constructs. Additionally, we specified the common rule anomalies such as conflict rules in formal properties such that the rules verification problem is transformed to a model checking one. These activity recognition rules in AMUPADH system are verified using this approach. Our case study found rule anomalies including non-reachable, redundant and conflict rules in the rule set with counterexamples revealing the execution paths leading to the errors.

Qualitative system analysis helps people to identify design flaws and improve system logic, but it does not give much intuition of how to improve the system consisting of many hardware components for the sake of both correctness and reliability. For example, if some inconsistency is found in the system, is it a better solution to replace a current sensor with a similar sensor of higher accuracy than improve the network reliability for less message loss. Naturally, quantitative analysis, especially probabilistic model checking techniques are chosen to tackle such problems.

Thus, in the third part of our work, we explore MDP-based reliability analysis techniques for improving the system design. MDP based formalism is chosen for modelling PvC systems because it has better support for modelling the non-deterministic system behaviours than other probabilistic modelling languages. In this work, a system model is manually constructed in MDP from the system design with reliability values and network topology provided by system engineers. Three groups of questions are explored which are "What is the overall system reliability with known reliability value of each nodes?", "To reach a certain overall system reliability, how reliable should the sensors/networks be?" and "Which node (could be a sensor or network device) affects the overall reliability the heaviest?". The AMUPADH system are modelled and verified using the model checker RaPid. Experiments

show surprising results that the overall system reliability is below 50%. It is also suggested that to improve the reliability of Wi-Fi network will be more efficient than replacing a sensor to improve the system reliability.

Overall, model checking techniques are found to be useful in analysing PvC systems. System flaws can be exposed in correctness analysis, erroneous rules are detected automatically and MDP-based reliability analysis can provide insights of improving system design. Analysing correctness of the system with timed requirements are also important and non-trivial. It faces the grand challenge of state space explosion. Yet, still there is lots of work to be done.

8.2 Future Challenges

Due to the various challenging issues in PvC systems, and the lack of comprehensive and complete solutions to them, we thus propose model checking techniques to be applied and specially tailored for these systems. In the following, we shall list the possible future research works. They are arranged in an order that looks like a research plan that gradually leads to the ultimate goal of formal and automated analysis of PvC systems.

8.2.1 Integrated Formal Modelling

As we explored in Chapter 4, there are different modelling languages for modelling different system features, for example, CSP# is perfect for modelling concurrent interactions and data processing in the system, Mobile Ambient are good at modelling location constraints, TCOZ is most suitable for specifying timing constraints of sensors and MDP is best for probabilistic behaviours. To model a large and hybrid system like PvC systems, any single one of these languages is not enough.

One possible solution is to design a powerful modelling language which integrates all these

useful language features. However, this solution produces a highly complex language that will create an enormous challenge for defining its semantics. Compass Modelling Language (CML) [88] is such a language proposed by Jim Woodcock and his team. It is a semantically heterogeneous language, with state-rich imperative constructs based on VDM, communication and concurrency based on CSP, object orientation with object references, and discrete time based on Timed CSP. In [87], they propose to use Unifying Theories of Programming (UTP) to describe semantic domains for CML.

An alternative solution is to integrate the heterogeneous models rather than the languages features. The idea is to create models of system components using the most suitable languages (e.g. CSP# for environment model, TCOZ for sensor models etc.) and glue/connect these models together using proper language constructs such as interfaces. This solution refers to the research problem of integrating heterogeneous models or describing architecture connection. Emerged in late 1990's, researchers has proposed Connectors [4] and Interface Automata [20] etc. as glues of component models. This approach does not have much limitation on the modelling languages of the system components. It is more focused on modelling/specifying the high-level interaction among system components.

Both of the two solutions seem feasible for modelling a PvC system considering its layered architecture and heterogeneous components. The former solution proposes a semantically open modelling language that new language features can be included if there is a need while the latter one is also flexible that it requires minimal effort in language design. However, for both of the approaches, defining the semantics of the resulting language remains a big challenge. We will explore along these directions in future.

8.2.2 Scalable Verification

In Chapter 5, the case study on AMUPADH evidenced the major bottleneck of applying model checking in complex systems- State Space Explosion. For a state space larger than 10^8 states, most explicit-state model checking tools will fail. However, real life applications usually have a much larger state space. It is a strong demand for us to explore more scalable approaches. We will discuss the possible future direction in the following.

State Space Reduction Technique The two most popular reduction techniques are partial order reduction and symmetry reduction. These techniques reduces state space respectively by pruning out excessive permutable execution orders of actions and by grouping symmetric states into equivalence classes and representing the equivalence classes with a representative state. In fact, there are such relations existing in PvC system models. For example, for a system shared by two patients who have the similar behaviours, the two actions of patient A enters the bedroom and patient B enters the bedroom are independent that either order of the two actions will have the same post state that both of the patients are in the bedroom. Thus partial order relation exists in such a case. As for symmetry relation, for two particular system states, a general reminder that not related to patient's identity is triggered in both states while one state with patient B is showering and patient A is sitting on bed and the other with patient A showering and patient B sitting on bed, these two states can be considered as symmetric and can be grouped.

Although these two techniques are effective, they are difficult to be applied for the reason that the partial order or symmetric relations are highly model dependent and needs to be discovered and defined prior to the implementation of the algorithms. Furthermore, these methods are only useful when there are lots of such relations in the system's state space. To find all the possible relations are highly non-trivial and require lots of work.

Compositional Verification Compositional verification on the other hand draws our attention. The idea was proposed dated back to the 1980s. Representative of such works are Assumed Guarantee by A. Pnueli in [61] and Interface Processes by E. M. Clarke et al. in [16]. They have contributed in decomposing global safety and liveness properties into local verifiable ones and deducing global property satisfaction from these local verification results.

Noticing the deadlock freeness verification experiments of Chapter 5, we believe that under some assumption, if all components are locally deadlock-free, 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. Furthermore, the general architecture of this type of systems suggests that there are no sharing recourses between components. The independence between system components further promotes that compositional verification could be a feasible solution in solving state space explosion problem in PvC system domain. In future, we shall research on this direction.

Although the research problem of compositional verification has established for 30 years, it remains a hot and tough topic. There are a few challenges for us. First of all, many of existing works explore process level composition under the same model while we are looking at model level composition under a model of heterogeneous models. Decomposing of properties must be carefully dealt with taking the coupling between component models into consideration. Furthermore, it is desirable to design automatic approaches for compositional verification in the context of large and complex PvC systems. It is a grand challenge if we are aiming to automatically verify a model of heterogeneous models.

Beside of the above problems, we also want to design a graphic modelling and simulation user interface. This is because, most designers and engineers from pervasive computing domains have limited background knowledge of formal methods. Visualisation of the modelling process and result simulation would be more useful and more fun.

Chapter 9

Bibliography

[1] Drools Verifier. http://community.jboss.org/wiki/DroolsVerifier. 6.3

[2] PAT: Process Analysis Toolkit. http://pat.comp.nus.edu.sg/. 2.2.2

[3] UPPAAL website. `http://www.uppaal.com/`. C, C.3

[4] Robert Allen and David Garlan. A formal basis for architectural connection. *ACM Trans. Softw. Eng. Methodol.*, 6(3):213–249, July 1997. 8.2.1

[5] Rajeev Alur and David L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, apr 1994. 2.2.4, 4.1.2

[6] Tobias Amnell, Gerd Behrmann, Johan Bengtsson, Pedro R. D;Argenio, Alexandre David, Ansgar Fehnker, Thomas Hune, Bertrand Jeannet, Kim G. Larsen, M. Oliver Mller, Paul Pettersson, Carsten Weise, and Wang Yi. Uppaal - now, next, and future. In *In Proc. MOVEP00, LNCS 2067*, pages 99–124. Springer, 2001. C

[7] Grigoris Antoniou. Rule-based activity recognition in ambient intelligence. In *Rule-Based Reasoning, Programming, and Applications*, page 1, 2011. 6

[8] Myrto Arapinis, Muffy Calder, Louise Denis, Michael Fisher, Philip D. Gray, Savas Konur, Alice Miller, Eike Ritter, Mark Ryan, Sven Schewe, Chris Unsworth, and Rehana Yasmin. Towards the verification of pervasive systems. *Electronic Communications of the EASST*, 22, 2009. 4, 4.3

[9] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking*. The MIT Press, 2008. 2.2.3, 7.2.2

[10] Matthias Baldauf, Schahram Dustdar, and Florian Rosenberg. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing*, 2(4):263–277, June 2007. 2.1.1

[11] Jit Biswas, Mounir Mokhtari, Jin Song Dong, and Philip Yap. Mild dementia care at home - integrating activity monitoring, user interface plasticity and scenario verification. In *Proceedings of the Aging friendly technology for health and independence, and 8th international conference on Smart homes and health telematics*, ICOST'10, pages 160–170, Berlin, Heidelberg, 2010. Springer-Verlag. 1.1.2, 2.1, 6

[12] Marius Bozga, Conrado Daws, Oded Maler, Alfredo Olivero, Stavros Tripakis, and Sergio Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *International Conf. on Computer Aided Verification (CAV)*. Springer, 1998. 2.2.4, C

[13] Luca Cardelli and Andrew D. Gordon. Mobile ambients. In *Proceedings of the First International Conference on Foundations of Software Science and Computation Structure*, pages 140–155, London, UK, 1998. Springer-Verlag. 4.3

[14] R. C. Cheung. A user-oriented software reliability model. *Software Engineering, IEEE Transactions on*, SE-6(2):118–125, March 1980. 7, 7.4

[15] EdmundM. Clarke and E.Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In Dexter Kozen, editor, *Logics of*

Programs, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer Berlin Heidelberg, 1982. 4.2.3

[16] E.M. Clarke, D.E. Long, and K. L. McMillan. Compositional model checking. In *Logic in Computer Science, 1989. LICS '89, Proceedings., Fourth Annual Symposium on*, pages 353–362, Jun 1989. 8.2.2

[17] CMU. Aura. http://www.cs.cmu.edu/ aura/. 2.1

[18] JBOSS Community. Drools expert user guide. http://docs.jboss.org/drools/release/5.2.0.CR1/drools-expert-docs/html/index.html. 6.1.1, 6.1.3

[19] Antonio Coronato and Giuseppe DE Pietro. Formal specification of wireless and pervasive healthcare applications. *ACM Trans. Embed. Comput. Syst.*, 10:12:1–12:18, August 2010. 4, 4.3, 6.3

[20] Luca de Alfaro and Thomas A. Henzinger. Interface automata. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-9, pages 109–120, New York, NY, USA, 2001. ACM. 8.2.1

[21] Jin Song Dong, Yuzhang Feng, Jing Sun, and Jun Sun. Context awareness systems design and reasoning. In *Proceedings of the Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation*, pages 335–340, Washington, DC, USA, 2006. IEEE Computer Society. 4, 4.3

[22] Kejun Du, Daqing Zhang, Xingshe Zhou, and Mossaab Hariz. Handling conflicts of context-aware reminding system in sensorised home. *Cluster Computing*, 14:81–89, March 2011. 4.2.2

[23] W. Keith Edwards and Rebecca E. Grinter. At home with ubiquitous computing: Seven challenges. In *Proceedings of the 3rd international conference on Ubiquitous Computing*, UbiComp '01, pages 256–272, London, UK, UK, 2001. Springer-Verlag. 2.1.2, 4

[24] Deborah Estrin, David Culler, Kris Pister, and Gaurav Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, January 2002. 4

[25] Swapna S. Gokhale. Architecture-based software reliability analysis: Overview and limitations. *IEEE Transaction on Dependable and Secure Computing*, 4(1):32–40, 2007. 7.4

[26] Swapna S. Gokhale, W. Eric Wong, Joseph Robert Horgan, and Kishor S. Trivedi. An analytical approach to architecture-based software performance and reliability prediction. *Perform. Eval.*, 58(4):391–412, 2004. 7

[27] Katerina Goseva-Popstojanova, Aditya P. Mathur, and Kishor S. Trivedi. Comparison of architecture-based software reliability models. In *ISSRE*, pages 22–33, 2001. 7

[28] Katerina Goševa-Popstojanova and Kishor S. Trivedi. Architecture-based approach to reliability assessment of software systems. *Performance Evaluation*, 45(2-3):179–204, 2001. 7.4

[29] Tao Gu, H.K. Pung, and Daqing Zhang. Peer-to-peer context reasoning in pervasive computing environments. In *Pervasive Computing and Communications, 2008. PerCom 2008. Sixth Annual IEEE International Conference on*, pages 406–411, 2008. 1.1.1, 4.3, 6

[30] Tao Gu, Zhanqing Wu, Liang Wang, Xianping Tao, and Jian Lu. Mining emerging patterns for recognizing activities of multiple users in pervasive computing. In *Mobile*

and Ubiquitous Systems: Networking Services, MobiQuitous, 2009. MobiQuitous '09. 6th Annual International, pages 1–10, 2009. 6

[31] Lin Gui, Jun Sun, Yang Liu, Yuanjie Si, Jin Song Dong, and Xinyu Wang. Combining model checking and testing with an application to reliability prediction and distribution. In Accepted, editor, *The International Symposium in Software Testing and Analysis (ISSTA 2013)*, Lugano, Switzerland, 15-20 July 2013. 2.2.3

[32] K. Havelund, A. Skou, K. G. Larsen, and K. Lund. Formal modeling and analysis of an audio/video protocol: an industrial case study using uppaal. In *RTSS '97: Proceedings of the 18th IEEE Real-Time Systems Symposium*, page 2, Washington, DC, USA, 1997. IEEE Computer Society. 2.2.4

[33] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:394–406, 1992. 4.2.3

[34] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21:666–677, August 1978. 2.2.2, 2.2.4, C.3

[35] C. Y. Huang and M. R. Lyu. Optimal testing resource allocation, and sensitivity analysis in software development. *IEEE Trans. Reliability*, 54(4):592–603, 2005. 7.4

[36] Anne Immonen and Eila Niemel. Survey of reliability and availability prediction methods from the viewpoint of software architecture. *Software and Systems Modeling*, 7(1):49–65, 2008. 7.4

[37] Edmund M. Clarke Jr., Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, 1999. 1, 2.2.1, 4

[38] Roope Kaivola, Rajnish Ghughal, Naren Narasimhan, Amber Telfer, Jesse Whittemore, Sudhindra Pandav, Anna Slobodová, Christopher Taylor, Vladimir Frolov, Erik

Reeber, and Armaghan Naik. Replacing testing with formal verification in intel coretm i7 processor execution engine validation. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 414–429, Berlin, Heidelberg, 2009. Springer-Verlag. 1

[39] M. Kwiatkowska, G. Norman, and D. Parker. PRISM 4.0: Verification of probabilistic real-time systems. In *CAV*, pages 585–591, 2011. 2.2.3, 4.1.2

[40] P. Lakey and A. Neufelder. *System and Software Reliability Assurance Notebook*. Rome Laboratory, Rome, New York, 1997. 7.4

[41] Kim G. Larsen, Paul Pettersson, and Wang Yi. Uppaal in a nutshell, 1997. 2.2.4

[42] Vwen Yen Lee, Yan Liu, Xian Zhang, Clifton Phua, Kelvin Sim, Jiaqi Zhu, Jit Biswas, Jin Dong, and Mounir Mokhtari. Acarp: Auto correct activity recognition rules using process analysis toolkit (pat). In *Impact Analysis of Solutions for Chronic Disease Prevention and Management*, volume 7251 of *Lecture Notes in Computer Science*, pages 182–189. Springer-Verlag, 2012. 1.3

[43] Antoni Ligeza and Grzegorz J. Nalepa. *Handbook of Research on Emerging Rule-Based Languages and Technologies: Open Solutions and Approaches*, chapter Rules Verification and Validation, pages 273–301. IGI Global,Hershey, 2009. 6.3

[44] Yan Liu. Formal Modelling and Verification of Pervasive Computing Systems. In Christine Choppy and Jun Sun, editors, *1st French Singaporean Workshop on Formal Methods and Applications (FSFMA 2013)*, volume 31 of *OpenAccess Series in Informatics (OASIcs)*, pages 61–67, Dagstuhl, Germany, 2013. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. 1.3

[45] Yan Liu, Lin Gui, and Yang Liu. Mdp-based reliability analysis of an ambient assisted living system. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014:*

Formal Methods, volume 8442 of *Lecture Notes in Computer Science*, pages 688–702. Springer International Publishing, 2014. 1.3

[46] Yan Liu, Xian Zhang, Jin Song Dong, Yang Liu, Jun Sun, J. Biswas, and M. Mokhtari. Formal analysis of pervasive computing systems. In *The 17th IEEE International Conference on Engineering of Complex Computer Systems*, pages 169 –178, 2012. 1.3

[47] Yan Liu, Xian Zhang, Yang Liu, JinSong Dong, Jun Sun, Jit Biswas, and Mounir Mokhtari. Towards formal modelling and verification of pervasive computing systems. In Ryszard Kowalczyk and Ngoc Thanh Nguyen, editors, *Transactions on Computational Collective Intelligence XVI*, Lecture Notes in Computer Science, pages 62–91. Springer Berlin Heidelberg, 2014. 1.3

[48] M. R. Lyu, S. Rangarajan, and A. P. A. van Moorsel. Optimal allocation of test resources for software reliability growth modeling in software development. *IEEE Trans. Reliability,*, 51(2):183–192, 2002. 7.4

[49] Brendan Mahony and Jin Song Dong. Blending object-z and timed csp: an introduction to tcoz. In *Proceedings of the 20th international conference on Software engineering*, ICSE '98, pages 95–104, Washington, DC, USA, 1998. IEEE Computer Society. 4.3

[50] Mark W. Maier. Architecting principles for systems-of-systems. *Systems Engineering*, 1(4):267–284, 1998. 2.1.2

[51] Will Marrero, Edmund Clarke, and Somesh Jha. Model Checking for Security Protocols. Technical report, Carnegie Mellon University, 1997. 4.2.1

[52] Roland Meyer, Johannes Faber, Jochen Hoenicke, and Andrey Rybalchenko. Model checking duration calculus: a practical approach. *Form. Asp. Comput.*, 20(4-5):481–505, June 2008. 1

[53] R. Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982. 2.2.2

[54] MIT. Oxygen. http://oxygen.csail.mit.edu/Overview.html. 2.1

[55] Yuji Nakagawa and Kyoichi Nakashima. A heuristic method for determining optimal reliability allocation. *IEEE Trans. Reliability*, R-26(3):156–161, 1977. 7.4

[56] Jürgen Nehmer, Martin Becker, Arthur Karshmer, and Rosemarie Lamm. Living assistance systems: an ambient intelligence approach. In *ICSE*, pages 43–50, 2006. 1

[57] TruongKhanh Nguyen, Jun Sun, Yang Liu, JinSong Dong, and Yan Liu. Improved bdd-based discrete analysis of timed systems. In Dimitra Giannakopoulou and Dominique Mry, editors, *FM 2012: Formal Methods*, volume 7436 of *Lecture Notes in Computer Science*, pages 326–340. Springer Berlin Heidelberg, 2012. 1.3

[58] Peter Csaba Olveczky and Stian Thorvaldsen. Formal modeling, performance estimation, and model checking of wireless sensor network algorithms in real-time maude. *Theor. Comput. Sci.*, 410:254–280, 2009. 4.1.2

[59] Amir Padovitz, Seng Wai Loke, and Arkady B. Zaslavsky. On uncertainty in context-aware computing: Appealing to high-level and same-level context for low-level context verification. In *IWUC*, pages 62–72, 2004. 1.1.1, 7

[60] R. Pietrantuono, S. Russo, and K. S. Trivedi. Software reliability and testing time allocation: An architecture-based approach. *IEEE Trans. Software Engineering*, 36:323–337, 2010. 7.4

[61] Amir Pnueli. In transition from global to modular temporal reasoning about programs. In KrzysztofR. Apt, editor, *Logics and Models of Concurrent Systems*, volume 13 of *NATO ASI Series*, pages 123–144. Springer Berlin Heidelberg, 1985. 8.2.2

[62] Alun D. Preece, Rajjan Shinghal, and Ada Batarekh. Principles and practice in verifying rule-based systems. *The Knowledge Engineering Review*, 7(02):115–141, 1992. 6.3

[63] Hung Keng Pung, Tao Gu, Wenwei Xue, Paulito P. Palmes, Jian Zhu, Wen Long Ng, Chee Weng Tang, and Nguyen Hoang Chung. Context-aware middleware for pervasive elderly homecare. *IEEE J.Sel. A. Commun.*, 27(4):510–524, May 2009. 1

[64] Martin L. Puterman. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1994. 2.2.3

[65] A. Ranganathan, J. Al-Muhtadi, and R. H. Campbell. Reasoning about uncertain contexts in pervasive computing environments. *IEEE Pervasive Computing*, 3(2):62–70, April 2004. 1.1.1, 7

[66] Genaína Rodrigues, David Rosenblum, and Sebastian Uchitel. Using scenarios to predict the reliability of concurrent component-based software systems. In *FASE'05*, pages 111–126, 2005. 7.4

[67] Daniel Ronzani. Ubiquitous computing and communication journal the battle of concepts: Ubiquitous computing, pervasive computing and ambient intelligence in mass media. *Ubiquitous Computing and Communication Journal*, 4, May 2009. 2.1

[68] Debashis Saha and Amitava Mukherjee. Pervasive computing: A paradigm for the 21st century. *Computer*, 36:25–31, March 2003. 2.1.2, 2.1.2

[69] Michele Sama, Sebastian Elbaum, Franco Raimondi, David S. Rosenblum, and Zhimin Wang. Context-aware adaptive applications: Fault patterns and their automated identification. *IEEE Trans. Softw. Eng.*, 36:644–661, September 2010. 4.3

[70] Michele Sama, David S. Rosenblum, Zhimin Wang, and Sebastian G. Elbaum. Model-based fault detection in context-aware adaptive applications. In *SIGSOFT FSE*, pages 261–271, 2008. 4.3

[71] M. Satyanarayanan. Pervasive computing: Vision and challenges. *IEEE Personal Communications*, 8:10–17, 2001. 2.1.2, 2.1.2

[72] Ling Shi and Yan Liu. Modeling and verification of transmission protocols: A case study on csma/cd protocol. In *Proceedings of the 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement Companion*, SSIRI-C '10, pages 143–149, Washington, DC, USA, 2010. IEEE Computer Society. 1.3

[73] William J. Stewart. *Introduction to the numerical solution of Markov chains*. Princeton University Press, 1994. 2.2.3

[74] H. Storf, M. Becker, and M. Riedl. Rule-based activity recognition framework: Challenges, technique and learning. In *PervasiveHealth*, pages 1–7, 2009. 1.1.1, 6

[75] Jun Sun, Yang Liu, Jin Song Dong, and Chunqing Chen. Integrating specification and programs for system modeling and verification. *Theoretical Aspects of Software Engineering, Joint IEEE/IFIP Symposium on*, 0:127–135, 2009. 2.2.2, 4, 4.1.1, 6

[76] Jun Sun, Yang Liu, Jin Song Dong, Yan Liu, Ling Shi, and Étienne André. Modeling and verifying hierarchical real-time systems using stateful timed csp. In *ACM Trans. Softw. Eng. Methodol.*, volume 22, pages 3:1–3:29, March 2013. 1.3, 4.1.2

[77] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. Pat: Towards flexible verification under fairness. In *Proceedings of the 21st International Conference on Computer Aided Verification*, CAV '09, pages 709–714, 2009. 1.1.2

[78] Jun Sun, Song Zheng Song, and Yang Liu. Model checking hierarchical probabilistic systems. In *12th International Conference on Formal Engineering Methods*, pages 388–403, 2010. 4.1.2

[79] Andrew S. Tanenbaum. *Computer networks: 2nd edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988. C

[80] Moshe Y. Vardi. Automatic verification of probabilistic concurrent finite state programs. In *Proceedings of the 26th Annual Symposium on Foundations of Computer Science*, SFCS '85, pages 327–338, Washington, DC, USA, 1985. IEEE Computer Society. 2.2.3

[81] Farn Wang, Rong-Shiung Wu, and Geng-Dian Huang. Verifying timed and linear hybrid rule-systems with red. In *SEKE*, pages 448–454, 2005. 2.2.4

[82] Xiao Hang Wang, Da Qing Zhang, Tao Gu, and Hung Keng Pung. Ontology based context modeling and reasoning using owl. In *Proceedings of the Second IEEE Annual Conference on Pervasive Computing and Communications Workshops*, PERCOMW '04, pages 18–22, 2004. 1.1.1, 4.3, 6

[83] Xiaohang Wang, Jin Song Dong, ChungYau Chin, SankaRavipriya Hettiarachchi, and Daqing Zhang. Semantic space: An infrastructure for smart spaces. *IEEE Pervasive Computing*, 3(3):32–39, July 2004. 1

[84] Mark Weiser. The computer for the 21st century. *Scientific American*, 265(3):66–75, January 1991. 1, 2.1, 4

[85] Wang Wen-Li, Wu Ye, and Chen Mei-Hwa. An architecture-based software reliability model. In *Dependable Computing, Proceedings. Pacific Rim International Symposium on*, pages 143–150, 1999. 7

[86] Jeannette M. Wing and Mandana Vaziri-Farahani. Model checking software systems: a case study. In *Proceedings of the 3rd ACM SIGSOFT symposium on Foundations of software engineering*, SIGSOFT '95, pages 128–139, New York, NY, USA, 1995. ACM. 1

[87] Jim Woodcock. Engineering utopia. In Cliff Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods*, volume 8442 of *Lecture Notes in Computer Science*, pages 22–41. Springer International Publishing, 2014. 8.2.1

[88] Jim Woodcock, Ana Cavalcanti, John S. Fitzgerald, Peter Gorm Larsen, Alvaro Miyazawa, and S. Perry. Features of cml: A formal modelling language for systems of systems. In *SoSE*, pages 445–450, 2012. 8.2.1

[89] Chang Xu and S. C. Cheung. Inconsistency detection and resolution for context-aware middleware support. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, ESEC/FSE-13, pages 336–345, New York, NY, USA, 2005. ACM. 4.3

[90] Chang Xu, S. C. Cheung, and W. K. Chan. Incremental consistency checking for pervasive context. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 292–301, New York, NY, USA, 2006. ACM. 4.3

[91] Sergio Yovine. Kronos: A verification tool for real-time systems. (kronos user's manual release 2.2). *International Journal on Software Tools for Technology Transfer*, 1:123–133, 1997. C, C.3

[92] Jiaqi Zhu, Vwen Yen Lee, Jit Biswas, Mounir Mokhtari, Thibaut Tiberghien, and Hamdi Aloulou. Context-aware reasoning engine with high level knowledge for smart home. In *PECCS*, pages 292–297, 2011. 6.1.1, 6.1.3

Appendix A

Operational Semantics of CSP#

where $e \in \Sigma$; $e_\tau \in \Sigma \cup \{\tau\}$; $x \in \Sigma \cup \{\checkmark\}$ and $* \in \Sigma \cup \{\tau, \checkmark\}$

$$\frac{}{(V, Skip) \xrightarrow{\checkmark} (V, Stop)} [\ skip\ ]$$

$$\frac{(V, P) \xrightarrow{e} (V', P'), e \in X}{(V, P \setminus X) \xrightarrow{\tau} (V', P')} [\ hide1\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P'), x \notin X}{(V, P \setminus X) \xrightarrow{x} (V', P' \setminus X)} [\ hide2\ ]$$

$$\frac{(V, P) \xrightarrow{e_\tau} (V', P')}{(V, P;\ Q) \xrightarrow{e_\tau} (V', P';\ Q)} [\ seq1\ ]$$

$$\frac{(V, P) \xrightarrow{\checkmark} (V', P')}{(V, P;\ Q) \xrightarrow{\tau} (V', Q)} [\ seq2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \square Q) \xrightarrow{x} (V', P')} [\ ch1\ ]$$

$$\frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \square Q) \xrightarrow{x} (V', Q')} [\ ch2\ ]$$

$$\frac{(V, P) \xrightarrow{\tau} (V', P')}{(V, P \square Q) \xrightarrow{x} (V', P' \square Q)} [\ ch3\ ]$$

$$\frac{(V, Q) \xrightarrow{\tau} (V', Q')}{(V, P \square Q) \xrightarrow{\tau} (V', P \square Q')} [\ ch4\ ]$$

$$\frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, P)}\,[\ non1\ ] \qquad \frac{}{(V, P \sqcap Q) \xrightarrow{\tau} (V, Q)}\,[\ non2\ ]$$

$$\frac{(V, P) \xrightarrow{x} (V', P')}{(V, P \| \ Q) \xrightarrow{x} (V', P' \| \ Q)}\,[\ int1\ ] \qquad \frac{(V, Q) \xrightarrow{x} (V', Q')}{(V, P \| \ Q) \xrightarrow{x} (V', P \| \ Q')}\,[\ int2\ ]$$

$$\frac{(V, P) \xrightarrow{\checkmark} (V', P'), (V, Q) \xrightarrow{\checkmark} (V', Q')}{(V, P \| \ Q) \xrightarrow{\checkmark} (V', P' \| \ Q')}\,[\ int3\ ] \qquad \frac{(V, P) \xrightarrow{*} (V', P')}{(V, P \triangle Q) \xrightarrow{*} (V', P' \triangle Q)}\,[\ inter1\ ]$$

$$\frac{(V, Q) \xrightarrow{e} (V', Q')}{(V, P \triangle Q) \xrightarrow{e} (V', Q')}\,[\ inter2\ ] \qquad \frac{(V, Q) \xrightarrow{\tau} (V', Q')}{(V, P \triangle Q) \xrightarrow{\tau} (V', P \triangle Q')}\,[\ inter3\ ]$$

Appendix B

A Complete List of Rules

Rule	Rule Condition	Action
0: Person entered Bedroom	rfidBedroom.state ≠ EMPTY	setPersonLocation(rfidBedroom, BEDROOM)
1: Person entered Shower Room	rfidShowerRoom.state ≠ EMPTY	setPersonLocation(rfidBedroom, SHOWERROOM)
2: Person sitting/-lying on Bed A	rfidBedA.state == personA && pressureBedA ≠ EMPTY	SendMsg: ACTIVITY.normal. (pressureBedB.state). CorrectBed. personA
3: Person sitting/-lying on Bed B	rfidBedB.state == personB && pressureBedB ≠ EMPTY	SendMsg: ACTIVITY.normal. (pressureBedB.state). CorrectBed. personB
4: personA sat on Bed A for too long (30mins)	pressureBedA.state == SITTING && pressureBedA.duration > 30	SendMsg: ACTIVITY. error. SitBedTooLong. personA
5: personB sat on Bed B for too long (30mins)	pressureBedB.state == SITTING && pressureBedB.duration > 30	SendMsg: ACTIVITY. error. SitBedTooLong. personB

6: Person lying on wrong bed (BedA)	rfidBedA.state != personA && rfidBedA.state ≠ EMPTY && pressureBedA.state ≠ LYING	SendMsg: ACTIVITY. error. LyingWrongBed. (rfidBedA.state)
7: Person lying on wrong bed (BedB)	rfidBedB.state != personB && rfidBedB.state ≠ EMPTY && pressureBedB.state ≠ LYING	SendMsg: ACTIVITY. error. LyingWrongBed. (rfidBedB.state)
8_1: PersonA is showering	shakeTap.state == UNSTATIONARY && shakeTap.duration < 40 && pirShowerRoom.state == FIRING && personA.location == SHOWERROOM	setShowerFlag(true); SendMsg: ACTIVITY. normal. Showering. PersonA
8_2: PersonB is showering	shakeTap.state == UNSTATIONARY && shakeTap.duration < 40 && pirShowerRoom.state == FIRING && personB.location == SHOWERROOM	setShowerFlag(true); SendMsg: ACTIVITY. normal. Showering. PersonB
9_1: PersonA is showering for too long (30 mins)	shakeTap.state == UNSTATIONARY && shakeTap.duration ≥ 120 && pirShowerRoom.state == FIRING && personA.location == SHOWERROOM	SendMsg: ACTIVITY. error. ShowerTooLong. personA
9_2: PersonB is showering for too long (30 mins)	shakeTap.state == UNSTATIONARY && shakeTap.duration ≥ 120 && pirShowerRoom.state == FIRING && personB.location == SHOWERROOM	SendMsg: ACTIVITY. error. ShowerTooLong. personB
10_1: PersonA is washing for too long and tap not off (15 mins)	shakeTap.state == UNSTATIONARY && shakeTap.duration ≥ 30 && pirShowerRoom.state == SILENT && personA.location == SHOWERROOM	SendMsg: ACTIVITY. error. ShowerNotOff. personA SendMsg: ACTIVITY. error. WanderingWashroom. personA

10_2: PersonB is washing for too long and tap not off (15 mins)	shakeTap.state == UNSTATIONARY && shakeTap.duration ≥ 30 && pirShowerRoom.state == SILENT && personB.location == SHOWERROOM	SendMsg: ACTIVITY. error. ShowerNotOff. personB SendMsg: ACTIVITY. error. WanderingWashroom. personB
11_1: PersonA is showering with no soap (15 mins)	shakeTap.state == UNSTATIONARY && shakeTap.duration ≥ 30 && shakeSoap.state == STATIONARY && shakeSopa.duration ≥ 30 && !soapFlag && pirShowerRoom == FIRING && personA.location == SHOWERROOM	setNoSoapEvent(true); SendMsg: ACTIVITY. error. ShowerNoSoap. personA
11_2: PersonB is showering with no soap (15 mins)	shakeTap.state == UNSTATIONARY && shakeTap.duration ≥ 30 && shakeSoap.state == STATIONARY && shakeSopa.duration ≥ 30 && !soapFlag && pirShowerRoom == FIRING && personB.location == SHOWERROOM	setNoSoapEvent(true); SendMsg: ACTIVITY. error. ShowerNoSoap. personB
12_1: PersonA just enters and wandering in washroom (15 mins)	shakeTap.state == STATIONARY && pirShowerRoom.state == FIRING && pirShowerRoom.duration ≥ 40 && !showerFlag && personA.location == SHOWERROOM	setWanderFlag(true); SendMsg: ACTIVITY. error. WanderingWashroom. personA
12_2: PersonB just enters and wandering in washroom (15 mins)	shakeTap.state == STATIONARY && pirShowerRoom.state == FIRING && pirShowerRoom.duration ≥ 40 && !showerFlag && personB.location == SHOWERROOM	setWanderFlag(true); SendMsg: ACTIVITY. error. WanderingWashroom. personB

13_1: PersonA forgot to switch off the tap or shower (15 mins)	shakeTap.state == UNSTATIONAY && shakeTap.duration ≥ 40 && soapFlag && !noSoapEvent && pirShowerRoom.state == FIRING && personA.location == SHOWERROOM	SendMsg: ACTIVITY. error. ShowerNotOff. personA
13_2: PersonB forgot to switch off the tap or shower (15 mins)	shakeTap.state == UNSTATIONAY && shakeTap.duration ≥ 40 && soapFlag && !noSoapEvent && pirShowerRoom.state == FIRING && personB.location == SHOWERROOM	SendMsg: ACTIVITY. error. ShowerNotOff. personB
14: Reset all sensor when non-activity in Washroom	shakeTap.state == STATIONARY && shakeSoap.state == STATIONARY && pirShowerRoom == SILENT && (washFlag == true || showerFlag == true || wanderFlag == true)	shakeTap.resetDuration(); shakeSoap.resetDuration(); setSoapFlag(false); setNoSoapEvent(false); setShowerFlag(false); setWanderFlag(false); SendMsg: ACTIVITY. normal. WashroomEmpty.*
15: Used soap	shakeSoap.state == UNSTATIONARY	setSoapFlag(true)
16_1: Remove ShowerNoSoap alert for personA	soapFlag == true && noSoapEvent == true && personA.location == SHOWERROOM	SendMsg: ACTIVITY. normal. ShowerWithSoap. personA
16_2: Remove ShowerNoSoap alert for personB	soapFlag == true && noSoapEvent == true && personB.location == SHOWERROOM	SendMsg: ACTIVITY. normal. ShowerWithSoap. personB

Appendix C

Case Study on A Transmission Protocol: CSMA/CD

Transmission protocols are one kind application of real-time systems, which are policies govern interactions among communication agents. They play an important part in computer networks and distributed systems. Many protocols have been successfully used, but they may suffer from some unexpected failures. The most common faulty in protocols is the occurrence of deadlock; others include loss of message, message destruction, and timeout. In the attempt of verifying real-time systems, we did a case study on a simple but typical transmission protocol CSMA/CD (Carrier Sense Multiple Access / Collision Detection) [79], which is widely used in Ethernet networks.

In the literature, Sergio Yovine in [91] used the tool KRONOS [12] to formally verify the CSMA/CD protocol. Timed automata is used to model the protocol which captures the system's time constraints in a explicit way. He used TCTL to verify important system properties such as reachability, bounded response etc., as well as using timed-abstracting equivalence means to compare a real time implementation of a system with an abstract and

untimed specification of it, verifying the correctness of system behaviours. A similar case study is done using UPPAAL model checker [6, 3] which is also based on timed automata. However, it is a tedious and error-prone task to explicitly set/reset clock variables in in timed automata, especially when the model size grows very large. Besides, the UPPAAL model of CSMA/CD is abel to reach a state where there is a deadlock which is infeasible in checking bounded most liveness properties.

In our case study, we use the modelling language STCSP to model this protocol for its richness of timed constructs and implicit clocks which are implemented to set/rest clock variables automatically while execution of the model. Implicit clocks have certain benefits that it can model the compositional timed systems, to satisfy high-level system requirements like deadline, timeout, timed interrupt, which can be composed sequentially, or in parallel. We also use timed refinement relationship to check system correctness like KRONOS using timed-abstracting technique. However, our refinement checking is to check whether an implementation satisfies a specification or not. It is different from KRONOS, which uses an extended version of branching time temporal logic named Computation Tree Logic(CTL) with time TCTL to do timed property checking. We also show our verification results of certain critical properties in our home-grown model checker PAT.

C.1 CSMA/CD: A Collision Detection Protocal for Local Area Network

In Ethernet network, several agents may be connected by a single bus. A problem arises that how to assign the usage of bus to only one of many agents who competes for. Carrier Sense, Multiple Access with Collision Detection (CSMA/CD) protocol describes one solution to this problem.

The simplified algorithm of CSMA/CD is shown in Fig. C.1. Roughly speaking, whenever

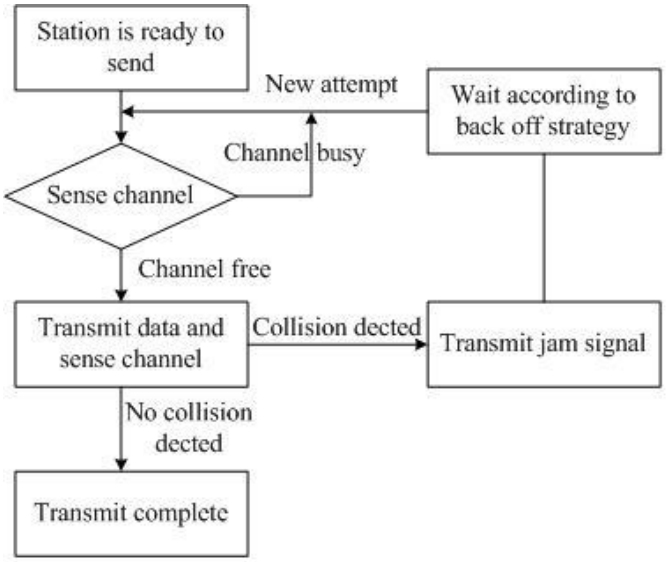


Figure C.1: Algorithm of CSMA/CD Protocol

an agent starts sending messages, it must first listen to the bus and wait for absence signal before transmitting. When the absence signal comes which means the bus is idle, the agent begins to transmit. If it detects a busy bus, it waits for a random amount of time before another try. As for the propagation time for message to travel from source node to the destination node via bus, an agent may listen to the bus to be idle while another agent is sending message before the message reaches any destination. Thus, collision occurs, then all of the agents are informed of this collision, and abort their transmission immediately. All transmitting messages are lost and all agents should compete for the bus again by waiting a random time.

C.2 Model for CSMA/CD protocol

As in real world, there are several important time parameters, such as different propagation time according to various materials of network wires. In order to better model the real

world protocol behaviour, we make the following assumptions. First we suppose that agents communicate in the 10Mbps Ethernet with a worst case propagation (denoted here by σ) for absence signal travel of 26 μsec. Additionally, we fix that messages have a fixed length of 1024 bytes, and the time for transmitting a complete message is assumed to be a constant time (denoted here by λ) 808 μsec, including propagation time. Besides, we don't model backoff strategy for retrying, we just assume that agent will retry within 2σ (52 μsec) time unit elapsed since the last step. Also, we make assumptions that no messages are lost during transmitting and there's no buffer for incoming messages at the agent side.

Based on the above assumptions, we then model the CSMA/CD protocol in the real-time system module in our PAT tool. The model for this protocol consists of two components, namely *Sender* (sending agent) and *Bus* (message transmitting channel). *Sender* and *Bus* communicate by synchronous events, so we define this communication by pair-wise synchronisation channels. In order to make all the variables and processes of this model to be clearly aware, we list all the related contents of this model with a simplified description, as illustrated in Table C.1.

Modeling Sender Behavior The behavior of component *Sender* is showed in Fig. C.2. *WaitFor* process models the behavior of sender i waiting for upper level messages to come. *Trans* process represents sender i completes transmitting messages within λ time unit or detects a collision within 2σ (52 μsec) time unit after its sending. *Retry* process expresses sender i wait for a 2σ (52 μsec) time unit to re-attempt.

Initially, the sender i is in *WaitFor* process. When a message arrives, one of the following transitions is executed. If the bus is not busy, the sender starts transmission. Otherwise, if bus is busy because another sender is already transmitting, it moves to retry state, or a collision is detected, it waits to retry. If a collision occurs while there is no message to send, the sender i remains in *WaitFor* state.

Category	Name	Description
Global Definition	N	Constant: number of senders
	channel newMess 0	Sender gets messages to send
	channel begin 0	Sender starts sending message
	channel busy 0	Sender senses a busy bus
	channel cd 0	Sender detects a collision
	channel end 0	Sender completes its transmission
Sender Behavior	WaitFor(i)	Sender i is waiting for a message from the upper level
	Trans(i)	Sender i is sending a message
	Retry(i)	Sender i is waiting to retry after detecting a collision or a busy bus
Bus Behavior	Idle	Bus is free, no sender is transmiting
	Active	One sender starts transmitting and is detecting collision
	Active1	One sender is transmitting messages, bus is busy
	Collision	Collision occurs and bus broadcasts the collision information to all senders

Table C.1: Components of CSMA/CD Model

In *Trans* process, sender i has two transitions, which is modelled as two external choices in PAT. If a collision is detected before 2σ time unit elapsed, the sender goes to *Retry* process. Otherwise, it terminates sending the message after exactly λ time unit, then it goes to the initial process.

When sender i is in *Retry* process, it makes a new step to resend messages before 2σ time unit elapsed since the last step. If the bus is idle, it will begin to transmit and moves to *Trans* state; If the bus is busy or receives a collision, it will still be in *Retry* state.

Modeling Bus Behavior The behavior of component Bus is showed in Fig. C.3. Initially, bus is in *Idle* process. When one sender starts sending its message, bus goes to *Active* process. If bus receives a signal that sender completes sending, it moves to idle state. Or after being in *Active* state for at least σ time unit, bus replies busy signal to any new attempt, which models the fact that the head of the message currently being sent has already

$$
\begin{aligned}
&WaitFor(i) = (cd?i \rightarrow WaitFor(i)) \\
&\qquad \Box(newMess!i \rightarrow ((begin!i \rightarrow Trans(i)) \\
&\qquad\qquad \Box(busy?i \rightarrow Retry(i)) \\
&\qquad\qquad \Box(cd?i \rightarrow Retry(i)))); \\
&Trans(i) = (cd?i \rightarrow Retry(i) within [0, 52]) \\
&\qquad \Box(atomic\{end!i \rightarrow Skip\} within [808, 808]; \\
&\qquad\quad WaitFor(i)); \\
&Retry(i) = (newMess!i \rightarrow ((begin!i \rightarrow Trans(i) within [0, 52]) \\
&\qquad\quad \Box(busy?i \rightarrow Retry(i) within [0, 52]) \\
&\qquad\quad \Box(cd?i \rightarrow Retry(i) within [0, 52])));
\end{aligned}
$$

Figure C.2: Model: the *Sender*

$$
\begin{aligned}
&Idle = newMess?i \rightarrow begin?i \rightarrow Active; \\
&Active = (end?i \rightarrow Idle) \\
&\qquad \Box(newMess?i \rightarrow \\
&\qquad\quad ((begin?i \rightarrow Collision)\ timeout[26] \\
&\qquad\quad \Box(busy!i \rightarrow Active1))); \\
&Active1 = (end?i \rightarrow Idle) \\
&\qquad \Box(newMess?i \rightarrow busy!i \rightarrow Active1); \\
&Collision = atomic\{BroadcastCD(0)\} within [0, 26];\ Idle;
\end{aligned}
$$

Figure C.3: Model: the *Bus*

propagated, then bus moves to *Active1* state. If another sender starts sending messages before σ time unit elapsed, bus moves to *Collision* state where it takes no more than σ time unit to broadcast collision to all senders. We use atomic process *BroadcastCD* shown in Fig. C.4 to broadcast collision to all senders. After that, bus moves to *Idle* state. When bus in *Active1* process, which means a sender has begun sending messages without collision, it will respond busy signal to all request senders until the sender completes transmitting, then bus moves to *Idle* state.

Composing CSMA/CD Protocol Model The whole system is executed by all senders and bus interleave with each other. The communication is implemented by the synchronous

$BroadcastCD(x) = \textbf{if}\ (x < N)\{$
$\quad (cd!x \rightarrow BroadcastCD(x+1))$
$\quad \Box$
$\quad (newMess?[i==x]i \rightarrow cd!x \rightarrow BroadcastCD(x+1))$
$\quad \}$
$\quad \textbf{else}\ \{$
$\quad\ Skip$
$\quad \};$

Figure C.4: Model: the *BroadcastCD process*

$CSMACD = (|||\ x : \{0..N-1\}@WaitFor(x)) \ |||\ Idle;$

Figure C.5: Model: the *CSMACD protocol*

channel between senders and bus. We model this as Fig. C.5

C.3 Verification and Experimental Results

Verification Properties In order to formally verify our model for CSMA/CD protocol is correct, we define several categories of properties to check whether it satisfies some properties. These properties in PAT can be categorized as *LTL-X Model Checking*, *Refinement Checking* and *Timed Refinement Checking*. In *LTL-X Model Checking*, properties are formulated using linear temporal logic formulae without next operator, which includes safety property and liveness property. *Refinement Checking* is to verify whether the system satisfies the property by showing a refinement relationship between the system and a model which models the property. The refinement relationship can be trace-refinement, stable failures refinement and failures/divergence refinement [34]. *Timed Refinement Checking* supports refinement checking between timed models, using implicit clocks and zone abstraction mechanism.

Deadlock Freeness (P0)

Informally, safety property states "bad things" never happen during the execution. Deadlock freeness is a safety property that has to be fulfilled so that it is always possible to move from one state to another. Deadlock freeness property in our model is defined as follows:

$\#assert\ CSMACD\ deadlockfree;$

Timed Divergence-free (P1)

If a process performs internal transitions and timed transitions forever without engaging any useful events, the process is said to be divergent. While the divergent system is undesirable, for it can give unbound timer, thus disallows timed refinement checking. Timed Divergence-free property in our model is defined as follows:

$\#assert\ CSMACD\ divergencefree < T >;$

Collision detection in a given bounded delay (P2)

Whenever two senders are simultaneously transmitting, a collision is detected in a bounded delay. In worst case, a sender can start sending at most σ time units after another sender, which means a collision occurs no more than σ time unit after two senders simultaneously transmit. And collision may take σ time units to be propagated. So a sender will detect a collision at most 2σ (52 μsec) after it starts transmitting.

Figure. C.6 shows a model that specifies this property. *Spec* shows that if event *begin*.0 occurs which means sender 0 begins transmitting, then *Constrained*1 happens. *Constrained*1 states if event *begin*.1 occurs thereafter which means sender 1 starts sending messages almost simultaneously, event *cd*.0 or *cd*.1 must occur within 52 time units, otherwise, no constraints apply, which is modeled as *Relaxed* process. In *Spec* process, if event *begin*.1 occurs and then followed by event *begin*.0, then *Constrained*2 happens. *Constrained*2 states if event

$$
\begin{aligned}
&Spec = (newMess.0 \rightarrow begin.0 \rightarrow Constrained1)\\
&\quad \Box(newMess.1 \rightarrow begin.1 \rightarrow Constrianed2)\\
&\quad \Box Relaxed;\\
&Constrained1 =\\
&\quad ((newMess.1 \rightarrow begin.1 \rightarrow\\
&\quad ((cd.0 \rightarrow Skip \Box cd.1 \rightarrow Skip) deadline[52]));\ Spec)\\
&\ \Box Relaxed;\\
&Constrained2 =\\
&\quad ((newMess.0 \rightarrow begin.0 \rightarrow\\
&\quad ((cd.0 \rightarrow Skip \Box cd.1 \rightarrow Skip) deadline[52]));\ Spec)\\
&\ \Box Relaxed;\\
&Relaxed =\\
&\quad (\Box x : \{2..N-1\}@(newMess.x \rightarrow begin.x \rightarrow Spec))\\
&\ \Box\\
&\quad (\Box x : \{0..N-1\}@((newMess.x \rightarrow\\
&\qquad\qquad (busy.x \rightarrow Spec \Box cd.x \rightarrow Spec))\\
&\qquad \Box(cd.x \rightarrow Spec)\\
&\qquad \Box(end.x \rightarrow Spec)));
\end{aligned}
$$

Figure C.6: Model: the *Collision detection in a given bounded delay*

begin.0 occurs thereafter which means sender 0 starts sending messages almost simultaneously, event *cd*.0 or *cd*.1 must occur within 52 time units, otherwise, it executes *Relaxed* process. In *Spec* process, if no constraints apply, it goes to *Relaxed* process. Our specification is to show whenever two senders send messages simultaneously, they will receive collision within 52 μsec since start transmitting.

In order to verify our model satisfies this property, we use timed refinement to check this requirement. Here, we define this in the following:

$$\#assert\ CSMACD\ refines < T >\ Spec;$$

Experimental Results Timed refinement checking allows us to verify Collision detection in a given bounded delay property which consists of timed transitions. We have experi-

Property	#Senders	Result	#States	#Transitions	Time /s
P0	4	Yes	787	1075	0.20
	5	Yes	2789	3847	0.60
	6	Yes	8851	12227	2.28
	7	Yes	26109	35991	8.43
	8	Yes	73123	100419	31.03
	9	Yes	196997	269319	108.69
	10	Yes	514915	700611	361.58
P1	4	Yes	787	1075	0.17
	5	Yes	2789	3847	0.66
	6	Yes	8851	12227	2.53
	7	Yes	26109	35991	9.79
	8	Yes	73123	100419	35.69
	9	Yes	196997	269319	123.24
	10	Yes	514915	700611	407.12
P2	4	Yes	787	1075	0.20
	5	Yes	2789	3847	0.90
	6	Yes	8851	12227	3.69
	7	Yes	26109	35991	14.74
	8	Yes	73123	100419	55.38
	9	Yes	196997	269319	196.35
	10	Yes	514915	700611	655.38

Table C.2: Experiment: Verification of CAMS/CD Protocol

mented CSMA/CD protocol on PAT for different number of senders. Table C.2 summarizes the verification results of properties. The experiment testbed is a PC running Windows XP3 within 2.33GHz Intel(R) core(TM)2 Duo CPU and 3.25GB memory.

From the table C.2, firstly we can see that the number of states, transitions and running time increase rapidly with the number of senders. Secondly, we can show that PAT is effective, for it can handle thousands of states in no more than 1000 seconds. The data on UPPAAL [3] or KRONOS [91] verifying the same models has been omitted from the table because KRONOS just model two senders, and model in UPPAAL [3] has a deadlock, it does not consider how to respond busy signal to request sender in multi-agents Ethernet networks. In fact, bus just broadcasts busy signal to all senders which cause the deadlock. Since our model does not present deadlock state, the more realistic modelling has brought

us more states then we can verify our model more correctly.