Verification of Timed Process Algebra and Beyond

Zhang Xian

(B.Sc. (Hons.), NUS)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2011

Acknowledgement

First and foremost, I am deeply indebted to my supervisor, Dr. Dong Jin Song, for his guidance, advice and encouragement throughout the course of my doctoral program. He has given me immense support both in various ways, and have also helped me stay on the track of doing research.

I am grateful to Dr. Joxan Jaffar and Dr. Andrew Santosa for their valuable suggestions and comments on my research works. I have special thanks to Dr. Sun Jun, Dr. Liu Yang, Dr. Hao Ping, Dr. Zhang Daqing, Dr. Qin Shengchao, Dr. Mikhail Auguston, Dr. Kenji Taguchi, etc for their research collaborations.

To my seniors, Dr. Li Yuanfang, Dr. Chen Chunqing, Dr. Sun Jing, Dr. Wang H. Hai and Feng Yuzhang - Thank you for your support and friendships through my Ph.D. study.

This study was in part funded by the project "Rigorous Design Methods and Tools for Intelligent Autonomous Multi-Agent Systems" and "Advanced Modeling Checking Systems" supported by Ministry of Education of Singapore and the project "Reliable Software Design and Development for Sensor Network Systems" supported by National University of Singapore Academic Research Fund and project "Model Checking System of Systems" supported by Temasek Defence Systems Institute and the project "Activity Monitoring and User interface Plasticity for supporting Ageing with mild Dementia at Home (AMUPADH)" supported by Agency for Science, Technology and Research (A*Star) Institute. The School of Computing also provided the finance for me to present papers in several conferences overseas.

Lastly, I wish to thank sincerely and deeply my parents Zhang Qiusheng and Li Yixia, who have taken care of me with great love in these years.

Contents

Summary

The design and verification of real-time systems are notoriously difficult problems. In this thesis, we study the modeling and verification of real-time systems using timed process algebra, particularly Timed CSP.

Timed CSP is an elegant and intuitive modeling language for real-time systems. It has been widely accepted and applied to a wide arrange of systems. However the verification support for Timed CSP is limited. The first part of the thesis is to develop a reasoning mechanism for Timed CSP by using Constraint Logic Programming (CLP) as underlying reasoning engine. Our approach starts with a systematic translation of the syntax of Timed CSP into CLP. Powerful constraint solver like CLP($\mathcal{R}$) is then used to prove traditional safety properties and beyond, e.g., reachability, deadlock-freeness, timewise refinement relationship, lower or upper bound of a time interval, etc. Counterexamples are generated when properties are not satisfied. Based on this translation, an interactive tool, named HORAE, which provides composing and reasoning of Timed CSP process descriptions is developed.

The second contribution of this thesis is the proposal of a formal language, named Timed Planning, for modeling real-time systems. Timed Planning extends Timed CSP with the capability of stating complicated timing behaviors. A Timed Planning model is made up of a hierarchical timed process and a set of constraints over processes, events and the data variables which are the requirements that the process should satisfy. Particularly, each process is associated with a set of localized timing/untiming requirements with keyword WHERE which can be specified in a compositional way. The full syntax and operational semantics of Timed Planning are formally defined. A reasoning mechanism for the Timed Planning is hence developed based on CLP by extending our reasoning engine HORAE. Feasibility checking and various property verification can be applied to check systems modeled in Timed Planning.

To show the usefulness of Timed Planning, we apply Timed Planning and HORAE to solve three different application domains. Firstly, we use Timed Planning to model classical job-

shop scheduling problems, in order to find a shortest execution in terms of elapsed time. In this case, the job-shop scheduling problem can be reduced to a problem of finding a complete execution (an execution that terminates) with the minimum execution time. In our work, Both deterministic and preemptive job-shop scheduling problems can be solved.

Secondly, security protocols are widely used for secure application-level data transport crossing distributed systems. Designing security protocols is notoriously difficult and error-prone. The new challenges raise when different timing aspects are required in the security protocol design, such as timestamps, delays, timeout and a set of timing constraints. We focus on using Timed Planning to accomplish the modeling and analyzing of timed security protocols. The use of explicit timing information allows us to specify security protocols with timestamps, timeout and retransmissions which can be naturally modeled using Timed Planning specification. In the timing analysis, we could verify timed non-injective agreement authentication property which can be easily extended to other authentication property verification. Besides, we can model timing requirements/constraints and verify other timed sensitive properties such as execution time of a protocol which is beyond the capability of existing approaches.

Thirdly, pervasive computing environments encompass a spectrum of computation and communication devices that seamlessly augment human thoughts and activities. They have been used to assist elders with mild dementia to improve their level of independence and quality of life through cognitive reinforcement. To support formal analysis, we propose to build a context-aware reminding framework for elders living at home using Timed Planning specification. Then we demonstrate the effectiveness of formal methods via modeling and verifying an integrated smart space reminding system for monitoring and assisting people with mild dementia in the nursing home.

Key words: **Timed Process Algebra, Formal Verification, Concurrent and Real-time Systems, Constraint Logic Programming, Refinement, Job-shop Scheduling Problem, Security Protocol, Dementia**

List of Figures

List of Tables

Chapter 1

Introduction

The design and verification of real-time systems are notoriously difficult problems, especially when systems often depend on quantitative timing. Modeling and verification of real-time systems are research topics which have important practical implication. Formal modeling and analysis of real-time systems are the trend to systematic and automatic detecting the design or implementation errors for complex systems.

1.1 Motivation and Goals

The correctness of many systems and devices in our modern society depends not only on the effects or results they produce but also on the time at which these results are produced. These real-time systems range from the anti-lock braking controller in automobiles to the vital-sign monitor in hospital intensive-care units. For example, when the driver of a car applies the brake, the anti-lock braking controller analyzes the environment in which the controller is embedded (car speed, road surface, direction of travel) and activates the brake with the appropriate frequency within fractions of a second. Both the result (brake activation) and the time at which the result is produced are important in ensuring the safety of the car, its driver, and passengers.

Recently, computer hardware and software are increasingly embedded in a majority of these real-time systems to monitor and control their operations. These computer systems are called real-time computer systems, or simply real-time systems. Unlike conventional, non-real-time systems, real-time systems are closely coupled with the environment being monitored and controlled. Examples of real-time systems include computerized versions of the braking controller and the vital-sign monitor, the new generation of airplane and spacecraft avionics, the planned Space Station control software, high-performance network and telephone switching systems, multimedia tools, virtual reality systems, robotic controllers, battery-powered instruments, wireless communication devices (such as cellular phones and PDAs), astronomical telescopes with adaptive-optics systems, and many safety-critical industrial applications. These real-time systems must satisfy stringent timing and reliability constraints in addition of functional correctness requirements.

In this thesis, our research focuses on the modeling and verification of real-time systems. In particular, we have tried to address four issues related to real-time systems: (i) proposing a formal language to model real-time systems, (ii) exploring efficient verification techniques to perform formal analysis of the models, (iii) implementing a toolkit to support effective real-time verification, and (iv) applying the proposed techniques in different domains.

1.1.1 Modeling Real-time systems

Real-time system modeling is of great importance and highly non-trivial. The choice of modeling language is an important factor in the success of the entire system analysis or development. The language should cover several facets of the requirements and the model should reflect exactly (up to abstraction of irrelevant details) an existing system or a system to be built. The language should have a semantic model suitable to study the behaviors of the system and to establish the validity of desired properties. Many languages have been proposed to model real-time systems, e.g., algebra of timed processes [79], timed CCS [108], timed CSP [88, 18], etc. The most popular language is timed automata [5, 71]. Timed automata are finite state automata equipped with clocks. They are powerful in de-

signing real-time models by explicitly manipulating clock variables. Real-time constraints are captured by explicitly setting or resetting clocks. A number of automatic verification support for timed automata based models have proven to be successful, e.g., UPPAAL [62], KRONOS [10].

Models based on timed automata often have a simple structure. For instance, the input models of the UPPAAL checker take the form of a network of timed automata with no hierarchy [62]. The advantage of a simple structure is that efficient verification is made feasible. Nonetheless, designing and verifying compositional real-time systems is becoming an increasingly important task due to the widespread applications and increasing complexity of real-time systems. High-level requirements for real-time systems are often stated in terms of *deadline*, *time out*, and *timed interrupt* [60, 32, 63]. In industrial case studies of real-time system verification, system requirements are often structured into phases, which are then composed in many different ways. Unlike Statecharts equipped with clocks [49] or timed process algebras [79, 108, 88], timed automata lack high-level compositional patterns for hierarchical design. Users often need to manually cast those terms into a set of clock variables with carefully calculated clock constraints. This process is tedious and error-prone. On the other hands, real-time system modeling based on timed process algebras often suffers from lack of language features (e.g., shared variables) or automated tool support.

One goal of this thesis is to design a modeling language for hierarchical real-time systems, which is sufficiently expressive, but is still subject to automatic verification.

Requirement Specification and Verification

Critical system requirements like safety, liveness and fairness play important roles in system specification, verification and development. Safety properties ensure that something undesirable never happens. Liveness properties state that something desirable must eventually happen.

In order to verify hierarchical real-time systems, more general specifications like refinement

relation are needed. In these cases, the requirement is modeled using an abstract model rather than a logic formula, which gives more expressive power. FDR (Failures-Divergence Refinement) [84] is the *de facto* refinement analyzer for CSP, which has been successfully applied in various domains. Based on the model checking algorithm presented in [84] and later improved with other reduction techniques presented in [86], FDR is capable of handling large systems. Nonetheless, when the system contains real-time constraints, new verification techniques and tools are required.

There are a few verification support for Timed CSP, e.g. the theorem proving approach documented in [11, 47], and the translation to UPPAAL models [25, 26]. The PAT model checker [1] is the first dedicated verification tool support for Timed CSP models by using dynamic zone abstraction. However, Timed CSP lacks support for stating system requirements which constrain all behavioral traces of given processes, for example, *deadline* and execution time of a process, time-related constraints among events which are common requirements for many real time systems. To increase the expressiveness of Timed CSP, current verification support becomes inadequate.

In this thesis, we want to explore the new approach for verification of powerful real-time system modeling language. To add to our understanding of the field, we aim to develop efficient model checker with the consideration of the above points.

1.2 Summary of Contributions

The main result of this thesis is a powerful modeling language for real-time systems together with the complete verification support. Several cases studies demonstrate the usefulness of our approach. The contributions of this thesis can be summarized as follows:

Event-based specification languages like the classic Communicating Sequential Process (CSP) of Hoare's [51] and its timed extension Timed CSP [82, 18], have been proposed for decades. Such specification languages are elegant and intuitive as well as precise. They have

been widely accepted and applied to a wide arrange of systems, including communication protocols, embedded systems, etc [85]. It is important that system specified using CSP or Timed CSP can be proved formally and even better if the proving is fully automated.

The first part of the thesis is building a reasoning mechanism for Timed CSP. Instead of building a specialized reasoning engine for Timed CSP from scratch, we choose one of powerful reasoning mechanisms, Constraint Logic Programming (CLP) [53, 54], as underlying reasoning engine. CLP has been successfully applied to model programs and transition systems for the purpose of verification [48, 56], showing that their approach outperforms the well-known state-of-art systems with higher efficiency. The XMC/RT system [37, 45] showed how a Timed Safety Automata (TSA) represented in CLP can be comparable in efficiency, by using a tabling logic programming framework. [4] employs a logic program transformation based approach for inductive verification of real-life parameterized protocols. Our approach starts with a systematic translation of the syntax of Timed CSP into CLP. Operational semantics are encoded into CLP in a systematic way. Powerful constraint solver like CLP($\mathcal{R}$) [55] is then used to prove traditional safety properties and beyond, e.g., reachability, deadlock-freeness, lower or upper bound of a time interval, etc. Counterexamples are generated when properties are not satisfied. Based on this translation, an interactive tool, named HORAE, which provides composing and reasoning of Timed CSP process descriptions is developed.

One of the main contribution of this thesis is the proposal of a formal language for modeling real-time systems, which is an extension of Timed CSP. We name this extension as Timed Planning. Timed Planning specification extends Timed CSP with the capability of stating complicated timing behaviors for processes and events to model and verify complex compositional real-time systems. A Timed Planning model is made up of a hierarchical timed process and a set of constraints over processes, events and the data variables which are the requirements that the process should satisfy. In this approach each process is associated with a set of localized timing/untiming requirements with keyword WHERE which can be specified in a compositional way. The full syntax and operational semantics of Timed

Planning language are formally defined. A reasoning mechanism for the Timed Planning is hence developed based on CLP by extending our reasoning engine HORAE. Both syntax and semantics of Timed Planning are formally translated into CLP. The operational semantics is encoded to CLP, where a set of global and local variables need to be captured during the execution. Feasibility checking and various property verification can be applied to check systems modeled in Timed Planning. Feasibility checking is necessary which helps users to debug the conflicts of the set of timing constraints specified in the systems.

In many application domains, we are interested in selecting, among all possible behaviors, one that *optimizes* some sophisticated performance measurement. We apply Timed Planning and HORAE to solve classical job-shop scheduling problems, where finding an optimal schedule corresponds to finding a shortest execution (in terms of elapsed time) in the Timed Planning. The observation underlying is that classical scheduling and resource allocation problems can be modeled naturally using Timed Planning whose runs correspond to feasible schedules. In this case, the job-shop scheduling problem can be reduced to a problem of finding a complete execution (an execution that terminates) with the minimum execution time. In our work, Both deterministic [58] and preemptive job-shop scheduling problems can be solved. We also apply our approach to handle the extended job-shop scheduling problems, where jobs can have more complex relations, such as a composition of operational behaviors with communications, and jobs with deadlines and relative timing constraints, which no other current work are able to support.

Another application of Timed Planning is to model and verify timed security protocols. Security protocols are widely used for secure application-level data transport crossing distributed systems. In general, designing security protocols is notoriously difficult and error-prone. Many protocols proposed in the literature and exploited in practice turned out to be awed, or their well-functioning was found to be based on implicit assumptions. Since the late 80s various approaches have been put forward for the formal verification of security protocols to overcome the problems of faulty implementations and hidden requirements. The new challenges raise when different timing aspects are required in the security protocol

design, such as timestamps, delays, timeout and a set of timing constraints. In the past years, there has been an increasing interest in the formal analysis of timed cryptographic protocols. However, there are few tool supports for modeling and analyzing security protocols with the capability of capturing various timing features. A particularly successful approach to analyze untimed security protocols is modeled using CSP [51] and verified by FDR model checker [87, 70]. Motivated by this approach, we focus on using Timed Planning to accomplish the modeling and analyzing of timed security protocols. Our approach is different from the previous approaches by taking the timing information into account. The use of explicit timing information allows us to specify security protocols with timestamps, timeout and retransmissions which can be naturally modeled using Timed Planning specification. In the timing analysis, we could verify timed non-injective agreement authentication property which can be easily extended to other authentication property verification [46]. We also propose to use the capability of Timed Planning to avoid such attacks without changing the original specifications of the protocols. Besides, we can model timing requirements/constraints and verify other timed sensitive properties such as execution time of a protocol which is beyond the capability of existing approaches.

Pervasive computing environments encompass a spectrum of computation and communication devices that seamlessly augment human thoughts and activities. Applications in this type of environments are often context-aware, using various kinds of context such as location and time to adapt to the evolving environments and provide smarter services. As the fast development of context-aware systems (e.g. new sensors, new application domains), critical system requirements like safety and liveness properties start to play more important roles. For example, smart system deployed in hospital shall never reach a state which hazards patients' lives. Therefore, formal methods are desirable in the design and development context-aware systems.

Pervasive computing techniques have been proposed to assist elders with mild dementia to improve their level of independence and quality of life through cognitive reinforcement. To support formal analysis, we propose to build a context-aware reminding framework for elders

living at home using Timed Planning specification. Then we demonstrate the effectiveness of formal methods via modeling and verifying an integrated smart space reminding system for monitoring and assisting people with mild dementia in the nursing home.

1.3 Thesis Outline and Overview

In this section, we briefly present the outline of the thesis and overview of each chapter.

Chapter 2 introduces background information on specification languages and tools used in the presented work. We firstly review CSP and Timed CSP with their syntax and semantics. Next we briefly introduce Constraint Logic Programming (CLP) and its related works in formal method domain.

Chapter 3 illustrates the verification system for Timed CSP using Constraint Logic Programming as underlying reasoning engine. We firstly show how the syntax of Timed CSP is formally translated into CLP. The operational semantics of each and every operator of Timed CSP are formally encoded into CLP. A reasoning tool for Timed CSP is presented in Section 3.5. Last but not least, we demonstrate the effectiveness of our approach through benchmark real-time systems.

Chapter 4 introduces an extended Timed CSP language, namely Timed Planning. We firstly define the syntax and operational semantics of Timed Planning formally. We further extend the verification system of Timed CSP to support Timed Planning using Constraint Logic Programming.

Chapters 5, 6 and 7 are devoted to applying Timed Planning in various applications to solve different problems. Chapter 5 applies Timed Planning in Job-shop scheduling problems where using underlying reasoning engine to find optimal scheduler for both preemptive and non-preemptive job-shop scheduling problems. Another piece of work has been done is that we apply Timed Planning to model and verify timed security protocols, which is

demonstrated in Chapter 6. In Chapter 7, we apply Timed Planning in pervasive computing, more specifically, the mild dementia care for elderly at both home and nursing home.

Lastly, Chapter 8 concludes this thesis, summarizes the main contributions and discusses possible future work directions.

1.4 Publications from the Thesis

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

The work in Chapter 3 was presented at *The* 8^{th} *International Conference on Formal Engineering Methods ICFEM'06 (December 2006)* [28]. The tool horae in Section 3.8 is published as a tool paper on *The* 13^{th} *International Symposium on Formal Methods FM'06 (November 2006)* [27]. The work in Chapter 4 was the basis for the paper published at *The* 4^{th} *International Conference on Secure Software Integration and Reliability Improvement (June 2010)* [112]. The work in Chapter 5 was presented at *The* 1^{st} *Workshop on International Conference on Secure Software Integration and Reliability Improvement (April 2010)* [111]. The work in Chapter 6 was accepted at *The* 4^{th} *International Conference on Secure Software Integration and Reliability Improvement (September 2010)* [112].

A paper including the modeling smart reminding system presented in Section 7 has been submitted for publication [98]. I also made contributions to other publications [24, 30, 31, 33, 99] which are remotely related this thesis.

For all the publications mentioned above, I have contributed substantially in both theory development and tool implementation.

A full list of the publications on which the thesis is based is included in the bibliography.

Chapter 2

Background Introduction

Real-time systems exhibit complex behaviors. System simulation and verification become more and more demanding as the complexity grows. It is highly desirable to have formal modeling languages to describe the real-time system rigourously. Compared with Timed Automata (with a flatten structure), timed process algebra is a more powerful way to modeling hierarchical real-times systems. Timed process algebra has a long history with various proposed formalisms. Examples include Timed CCS [108], Timed Communicating Sequential Processes (Timed CSP) [89], Timed petri nets [77] and so on. In this thesis, we focus on Timed CSP for its expressive power and wide usage.

Constraint logic programming (CLP) [54] is a form of constraint programming, in which logic programming is extended to include concepts from constraint satisfaction. In this chapter, we briefly introduce the basic concepts and conventions of CLP, which are used in the rest of the thesis.

The remainder of the chapter is organized as follows. Section 2.1 gives an introduction to CSP. The timed extension of CSP is explained in Section 2.2 with formal syntax and semantics. Section 2.3 presents the fundamental concepts of CLP. Section 2.4 introduces the model checker PAT.

2.1 Communicating Sequential Processes (CSP)

In computer science, Communicating Sequential Processes (CSP) [52] is a formal language for describing patterns of interaction in concurrent systems. It is a member of the family of mathematical theories of concurrency known as process algebras, or process calculi. CSP was first described in [51] by C. A. R. Hoare, but has evolved substantially since then. FDR (Failures-Divergence Refinement) [84] is the *de facto* refinement analyzer for CSP, which has been successfully applied in industry as a tool for specifying and verifying the concurrent aspects of a variety of different systems. The theory of CSP is still the subject of active research, including work to increase its range of practical applicability. CSP has passed the test of time. It has been widely accepted and influenced the design of many recent programming and specification languages including Ada [40], occam [75], Concurrent ML [83], BPEL4WS [81], TCOZ [74, 72, 25, 73], OZTA [24, 30] etc.

As its name suggested, CSP allows the description of systems in terms of component processes that operate independently, and interact with each other solely through message-passing communication. However, the "Sequential" part of the CSP name is now something of a misnomer, since modern CSP allows component processes to be defined both as sequential processes, and as the parallel composition of more primitive processes. The relationships between different processes, and the way each process communicates with its environment, are described using various process algebraic operators. Using this algebraic approach, quite complex process descriptions can be easily constructed from a few primitive elements.

CSP is a formal specification language where processes proceed from one state to another by engaging in events. Processes may be composed by using operators which require synchronization on events, i.e., each component must be willing to participate in a given event before the whole system makes the transition. Synchronous communication, rather than assignments to shared state variables, is the fundamental means of interaction between agents. In the following, we present the syntax of CSP process and informal semantics. For any process P in CSP language, it can be defined using following syntax.

$$
\begin{array}{lll}
P = \mathit{Stop} \mid \mathit{Skip} & & \text{– primitives} \\
\mid\ e \rightarrow P & & \text{– event prefixing} \\
\mid\ \mathit{ch}!\mathit{exp} \rightarrow P \mid \mathit{ch}?x \rightarrow P & & \text{– channel communications} \\
\mid\ P \setminus X & & \text{– hiding} \\
\mid\ P\ ;\ Q & & \text{– sequential composition} \\
\mid\ P\ \Box\ Q \mid P \sqcap Q & & \text{– choice operators} \\
\mid\ [b]P & & \text{– state guard} \\
\mid\ P \parallel Q & & \text{– parallel composition} \\
\mid\ P \mid\mid\mid Q & & \text{– interleave composition} \\
\mid\ P \triangledown Q & & \text{– interrupt} \\
\mid\ \mathit{ref}(Q) & & \text{– process reference}
\end{array}
$$

where P, Q are processes, e is a name representing an event with an optional sequential program[1] *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 communicates nothing, also called deadlock. $\mathit{Skip} = \checkmark \rightarrow \mathit{Stop}$, where $\checkmark$ is the special event of termination. Event prefixing $e \rightarrow P$ performs e and afterwards behaves as process P. Hiding process $P \setminus X$ makes all occurrences of events in X not to be observed or controlled by the environment of the process. 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[2]. On the contrast, internal choice $P \sqcap Q$ is solved non-deterministically. Process $[b]P$ waits until condition b becomes true and then behaves as P. Note that $[b]P$ does not block and will be dropped in choice operators if other choices are selected. 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 $P \parallel Q$ synchronizes common events in the alphabets of P and Q excluding *non-communicating events*. In contrast, process $P \mid\mid\mid Q$ runs all processes independently (except for communication through shared variables and synchronous channels[3]). Process $P \triangledown Q$ behaves as P

[1] The grammar rules for the sequential program can be found in PAT user manual.

[2] In CSP, symbol τ is introduced as the internal action in the operational semantics only. Any event other than τ is called visible event.

[3] Note that in original CSP, $P \mid\mid\mid Q$ does not allow communication through shared channels.

until the first occurrence of a visible event from Q. A process expression may be given a name for referencing. Recursion is supported by process referencing.

2.2 Timed CSP

2.2.1 Syntax of Timed CSP

Hoare's CSP [51] is an event based notation primarily aimed at describing the sequencing of behavior within a process and the synchronization of behavior (or *communication*) between processes. Timed CSP extends CSP by introducing a capability to quantify temporal aspects of sequencing and synchronization. Inherited from CSP, Timed CSP adopts a symmetric view of process and environment. Events represent a cooperative synchronization between process and environment. Both process and environment may control the behavior of the other by *enabling* or *refusing* certain events and sequences of events.

Definition 1 (Timed CSP) *A Timed CSP process is defined by the following syntax,*

$$
\begin{array}{rcl}
P & ::= & \textsc{Stop} \mid \textsc{Skip} \mid \textsc{Run} \mid e \stackrel{t}{\rightarrow} P \mid e : E \rightarrow P(e) \mid e@t \rightarrow P(t) \\
& \mid & P_1 \,\Box\, P_2 \mid P_1 \sqcap P_2 \mid P_1 \;{}_X||_Y\; P_2 \mid P_1 \,|[\, X \,]|\, P_2 \mid P_1 \,|||\, P_2 \\
& \mid & P_1;\; P_2 \mid P_1 \,\triangledown\, P_2 \mid P1 \stackrel{d}{\triangleright} P2 \mid \textsc{Wait}\; d \mid P_1 \,\triangledown_d\, P_2 \\
& \mid & \mu\, X \bullet P(X)
\end{array}
$$

$\textsc{Run}_\Sigma$ is a process always willing to engage any event in Σ. Stop denotes a process that deadlocks and does nothing. A process that terminates is written as Skip. A process which may participate in event e then act according to process description P is written as $e@t \rightarrow P(t)$. The (optional) timing parameter t records the time, relative to the start of the process, at which the event e occurs and allows the subsequent behavior P to depend on its value. The process $e \stackrel{t}{\rightarrow} P$ delays process P by t time units after engaging event e. The external choice operator, written as $P \,\Box\, Q$, allows a process of choice of behavior according to what events are requested by its environment. Internal choice represents variation in behavior determined by the internal state of the process. The parallel composition of processes P_1

and P_2, synchronized on common events of their alphabets X, Y (or a common set of events A) is written as $P_1 \ {}_X||_Y \ P_2$ (or $P_1 \,|[\, A \,]|\, P_2$). The sequential composition of P_1 and P_2, written as $P_1;\ P_2$, acts as P_1 until P_1 terminates by communicating a distinguished event $\checkmark$ and then proceeds to act as P_2. The interrupt process $P_1 \ \triangledown \ P_2$ behaves as P_1 until the first occurrence of event in P_2, then the control passes to P_2. The timed interrupt process $P_1 \ \triangledown_d \ P_2$ behaves similarly except P_1 is interrupted as soon as d time units have elapsed. A process which allows no communications for period d time units then terminates is written as WAIT d. The timeout construct written as $P_1 \overset{d}{\triangleright} P_2$ passes control to an exception handler P_2 if no event has occurred in the primary process P_1 by some deadline d. Recursion is used to give finite representation of non-terminating processes. The process expression $\mu\, X \bullet P(X)$ describes processes which repeatedly act as $P(X)$. The detailed illustration of each process can be found in [89].

2.2.2 Semantics of Timed CSP

The semantics of a Timed CSP process is precisely defined either by identifying how the process may evolve through time or by engaging in events (i.e., the operational semantics defined in [90]) or by stating the set of observations, e.g., traces, failures and timed failures (i.e., the denotational semantics as defined in [17]). In this work, Timed CSP is used to specify interactive timed tasks.

In general, the behavior of a process at any point in time may be dependent on its internal state and this may conceivably take an infinite range of values. It is often not possible to provide a finite representation of a process without introducing some notation for representing this internal state. The approach adopted by Timed CSP is to allow a process definition to be parameterized by state variables. Thus a definition of the form $P(x)$ represents a family of definitions, one for each possible value of x.

Example 2.2.1 (Timed Vending Machine) A user may insert some coins and then make a choice between coffee or tea. Once the choice is made, the vending machine dispatches the

corresponding drink. Or the user may ask the machine to release the coins and walk away. If the user idles more than 10 seconds after the coin is inserted, the machine will release the coins.

$$\begin{aligned} TVM \mathrel{\widehat{=}} \mu\, X \bullet\ & coin \rightarrow ((reqrelease \rightarrow release \overset{2}{\rightarrow} X) \\ & \Box\ (coffee \overset{3}{\rightarrow} dispatchcoffee \rightarrow X) \Box\ (tea \overset{2}{\rightarrow} dispatchtea \rightarrow X)) \\ & \overset{10}{\triangleright}\ (release \rightarrow X) \end{aligned}$$

□**end**

2.3 Constraint Logic Programming

Constraint Logic Programming (CLP) [54] began as a natural merger of two declarative paradigms: constraint solving and logic programming. This combination helps to make CLP programs both expressive and flexible, and in some cases, more efficient than other kinds of logic programs. The CLP scheme defines a class of languages based upon the paradigm of rule-based constraint programming. CLP($\mathcal{R}$) [55] is an instance of this class, which is used in this thesis. We present some preliminary definitions about CLP in this section.

Definition 2 (Atom, Rule and Goal) *An atom is of the form $p(\tilde{t})$, where p is a user defined predicate symbol and $\tilde{t}$ is a sequence of terms '$t_1, t_2, \ldots, t_n$'. A rule is of the form $A :- \tilde{B}, \Psi$ where the atom A is the head of the rule, and the sequence of atoms $\tilde{B}$ and the constraint Ψ constitute the body of the rule. A goal has exactly the same format as the body of the rule of the form $? - \tilde{B}, \Psi$. If $\tilde{B}$ is an empty sequence of atoms, we call this a (constrained) fact. All goals, rules and facts are terms.*

The *universe of discourse* $\mathcal{D}$ of CLP program is a set of terms, integers, and lists of integers. A *constraint* is written using a language of functions and relations. They are used in two ways, in the basic programming language to describe expressions and conditions, and in user assertions, as defined below. In this work, we will not define the constraint language

explicitly, but invent them on demand in accordance with our examples. Thus the terms of our CLP programs include the function symbols of the constraint language. A *ground instance* of a constraint, atom and rule is defined in obvious way. A *ground instance* of a constraint is obtained by instantiating variables therein from $\mathcal{D}$. The *ground instances* of a goal G, written $[\![G]\!]$ is the set of ground atoms obtained by taking all the true ground instances of G and then assembling the ground atoms therein into a set. We write $G_1 \models G_2$ to mean that for all groundings θ of G_1 and G_2, each ground atom in $G_1\theta$ appears in $G_2\theta$.

Let $G = (B_1, ..., B_n, \Psi)$ and P denote a goal and program respectively. Let $R = A :- C_1, ..., C_m, \Psi_1$ denote a rule in P, written so as none of its variables appear in G. Let $A = B$, where A and B are atoms, be shorthand for equations between their corresponding arguments. A *reduct* of G using R is of the form

$$(B_1, ..., B_{i-1}, C_1, ..., C_m, B_{i+1}, ..., B_n, B_i = A \wedge \Psi \wedge \Psi_1)$$

provided $B_i = A \wedge \Psi \wedge \Psi_1$ is satisfiable. A *derivation sequence* is a possibly infinite sequence of goals $G_0, G_1, ...$ where $G_i, i > 0$ is a reduct of G_{i-1}. If there is a last goal G_n with no atoms, notationally $(\Box, \Psi)$ and called a *terminal goal*, we say that the derivation is a *successful* and that the *answer constraint* is Ψ. A derivation is ground if every reduction therein is ground.

Example 2.3.1 (Fibonacci) The following is typical CLP program:

fib$(0, 1)$. *fib*$(1, 1)$.
fib$(N, X1 + X2) :- N > 1, fib(N - 1, X1), fib(N - 2, X2)$.

A relation *fib(N, X)* is defined, where X is the factorial of N, denoted as $X = N!$. There are three atoms for the relation *fib(N, X)*, where the first two atoms are *fact*s and the last one is a *rule*. We want to find the number whose fibonacci is greater than 3 but less than 8, by executing the goal $? - F > 3, F < 8, fib(N, F)$. which returns $N = 4, F = 5$. □**end**

CLP has been successful as a programming language, and more recently, as a model of executable specifications. There have been numerous works which use CLP to model system

modeling or programs and which use an adaptation of the CLP proof system for proving certain properties [57, 29]. In this work, we follow this trend and use existing powerful constraint solvers for mechanized Timed Planning.

2.3.1 CLP and Reasoning

Constraint Logic Programming has been used to model programs and transitions systems for purpose of verification problems. In particular, it has been used to model Timed Safety Automata (TSA) [50]. The main advantage of using CLP pertain to expressiveness. For example, [48] demonstrates the proof of some standard properties, as well as properties such timed bounds between important events, on a CLP representation of TSA.

[56] presented a CLP proof method for well-known state-of-art systems - Timed Automata. They started with a systematic translation of TSA into CLP and then use a new CLP inference method for proving assertions. They claimed that their assertion language can specify important properties beyond traditional safety properties, one important of which is the *symmetric*. They also gave a demonstration showing that their improvements over the Timed Automata model checker - Uppaal [62].

Moreover, (Constraint) Logic Programming has been used in formal methods area, for example, [106] uses Logic Programming Techniques for animating Z which could be used for detecting errors in a formal specification.

2.4 Verification and Process Analysis Toolkit (PAT)

In this section, we introduce the verification tool for Timed CSP. Due to the expressive power, verification of Timed CSP is a difficult task with minimum tool support. To the best of our knowledge, there are few verification support for Timed CSP, e.g. the theorem proving approach documented in [11, 47], the translation to Uppaal models [25, 26] and the approach based on constraint solving [28]. Among them, the most established tool is

Process Analysis Toolkit (PAT) [1, 66, 95, 68, 64, 101, 113, 103]. PAT is a self-contained toolkit to analyze real-time systems, which supports system modeling, animated simulation and automatic verification (based on advanced model checking techniques like dynamic zone abstraction [99]).

PAT's modeling language [99, 94, 102, 69, 97] is an extension of Timed CSP with mutable variables and data structures (e.g., arrays, stacks or arbitrary data types), synchronous or asynchronous channels, etc. The language adopts a dense-time semantics, where the clock values are rational numbers. Hence, there may be infinitely many transitions between any two time points. To offer efficient verification support, a fully automated abstraction technique is developed to build an abstract finite state system from the (infinite) model. It has been prove that the abstraction has finite state and is subject to model checking [99]. Further, it weakly bi-simulates the concrete model and, therefore, it may perform sound and complete safety property checking, Linear Temporal Logic (LTL) [91] model checking, refinement checking upon the abstraction.

Figure 2.1 shows the architecture design of PAT with four components, namely the editor, the parser, the simulator and verifiers. 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. Due to infinite timed transitions, abstraction is applied to the input models so that a finite state abstract model is yielded internally. 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. The simulator is also used to visualize Büchi automata generated from the negation of LTL assertions. Most importantly, PAT implements several verifiers catering for checking deadlock-freeness, reachability, LTL properties with fairness assumptions [], refinement checking [23, 65] and etc. To achieve good performance, advanced optimization techniques are implemented, e.g., partial order reduction, process counter abstraction [100], parallel model checking [67], etc. All the verification algorithms perform on-the-fly exploration of the state space [110]. If any counterexample is identified during the exploration,

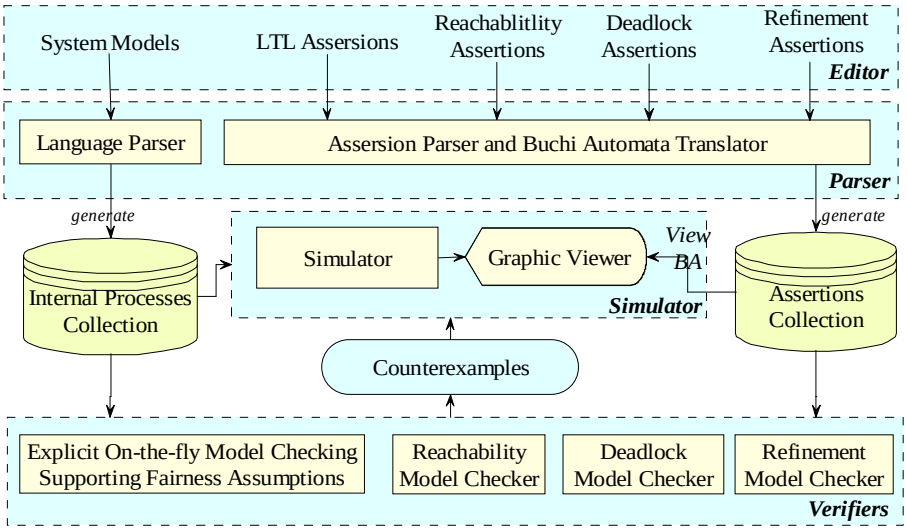

Figure 2.1: Architecture Design

then it can be animated in the simulator for the purpose of debugging. The correctness of PAT is established by using various test cases and user feedback. In additional, PAT is verified by performing model checking on its source code [93].

Real-Time System Modeling in PAT

In PAT, a system model is composed of multiple elements, i.e. constants, global variables/channels, a set of timed process definitions, a set of assertions, etc. The process definitions identify the computational logic of a system. A timed process P (hereafter process) can be defined using a rich set of process constructs (similar to CSP processes). Furthermore, a number of timed process constructs (similar to Timed CSP processes) can be used to capture common real-time system behavior patterns. For example, let d be an rational number. Process $Wait[d]$ idles for d time units. In process P $timeout[d]$ Q, the first observable event of P shall occur before d time units elapse. Otherwise, Q takes over control 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 over control. Process P *waituntil*$[d]$ is constrained to finish only after d time units have elapsed since the process starts. If P terminates early, then it idles until exactly d time units have elapsed. Process P *deadline*$[d]$ constrains P to terminate before d time units. In this setting, clock variables are made implicit and hence they cannot be compared with each other directly, which potentially allows efficient clock manipulation and hence system verification. In real-time systems, requirements are often structured into phases, which is hierarchical in nature [78]. PAT modeling language is hierarchical and uses implicit clocks, hence the modeling process is much simpler without complicated clock calculations. The complete language syntax can be found in PAT's user manual. The formal operational semantics can be found in [99].

Abstraction and Verification

Model checking requires a finite state system model. Hence, we assume that all variables have finite domains and the process forbids unbounded non-tail recursion. However, the number of system states (and hence the transition system) is still infinite because of our dense-time semantics. PAT adopts a zone abstraction [99] to build an abstract system. Different from zone abstraction applied to Timed Automata [21], PAT dynamically creates/deletes a set of clocks to precisely encode the timing requirements. A *zone* is the maximal set of clock valuations satisfying a set of primitive clock constraints. A primitive constraint on a clock is of the form $tm \sim d$ where tm is a timer, d is a constant and $\sim$ is $\geq$, $=$, or $\leq$. Because clocks are implicit, clock readings cannot be compared directly. In order to support efficient verification, PAT uses difference bound matrices (DBM) [21] as an equivalent representation for the zone.

To perform verification on the original systems, PAT shows that the abstract transition system is equivalent to the original transition system. It is shown that our zone abstraction is sound and complete with respect to the following three properties using a specialized bi-simulation relationship [99].

LTL Model Checking In this setting, the properties are linear temporal logic (LTL) formulae, constituted by propositions on global variables and events. Notice that no clocks are allowed in the property. In order to reflect model checking results on the abstract transition system to the original system with respect to LTL formulae, it is need to establish that the abstract transition system is equivalent to the concrete one with respect to LTL-X formulae. The idea is to show stutter equivalence between traces of the abstract system and the original system [99]. To verify the LTL formulae, PAT adopts the automata-based on-the-fly verification algorithm [96], i.e., by firstly translating a formula to a Büchi automaton and then check emptiness of the product of the system and the automaton.

Refinement Checking In this setting, PAT investigates an alternative verification approach. That is, to verify whether the system satisfies the property by showing a refinement relationship between the system and a model which models the property. In order to check refinement between two (timed) models, zone abstraction must be applied to both models. In [99], it proves that it is sound and complete to show stable failures refinement between the two abstraction transition systems in order to show failures refinement between the two corresponding original models. The refinement relationship is verified using an on-the-fly simulation checking approach.

Chapter 3

Encoding Timed CSP in CLP

Timed CSP is a powerful modeling language for real-time systems. However the verification support is limited. The first piece of work we have done is building a reasoning mechanism for Timed CSP. We choose CLP as the underlying reasoning support for Timed CSP, which can provide a simple and intuitive way of representing real timing aspects of Timed CSP and a flexible way of specifying interesting properties, such as deadlock-freeness, trace related properties, time related properties, and etc.

This chapter is organized as follows. In Section 3.1, we start with translating syntax of Timed CSP into CLP. Section 3.2 encodes the operational semantics of Timed CSP into CLP, by capturing both event transition and timed transition of every operator defined in Timed CSP (refer to Section 2.2). Section 3.3 encodes the denotational semantics of Timed CSP into CLP. In Section 3.4, we demonstrate how to verify the properties of the Timed CSP processes encoded in CLP based on the operational semantics. In Section 3.5, a prototype tool HORAE is developed to support the verification of the Timed CSP with the techniques we developed in this chapter. Section 3.6 demonstrate our approaches using three examples with experimental results. Section 3.7 concludes this chapter.

3.1 Timed CSP Specification in CLP

3.1.1 Syntax Encoding

In this section, we show how to encode every operator of Timed CSP defined in Section 2.2 into CLP terms. All operators of the extended specification which inherited from Timed CSP are encoded into CLP rules in a compositional way. A translation library for all operators is built. For example:

- $a \rightarrow$ SKIP : `eventprefix(a, skip)`
- $a \stackrel{t}{\rightarrow}$ SKIP : `delay(a, skip, t)`
- $P_1 \mid\mid\mid P_2$: `interleaving(P1,P2)`
- $P_1;\ P_2$: `sequential(P1,P2)`

In our library, *eventprefix*(A, P) is defined to denote a process $A \rightarrow P$. *delay*(A, P, T) is the CLP form of operator $A \stackrel{T}{\rightarrow} P$ in Timed CSP. *interleaving*$(P1, P2)$ is to represent operator $P_1 \mid\mid\mid P_2$ where $P1$ and $P2$ are the CLP formate of process P_1 and P_2. Relations *sequential*$/2$[1] is to represent a sequential operator ";". We define a library of rules for all Timed CSP processes in CLP, which is shown in Table 3.1.1.

The very initial step of our work is the syntax encoding of Timed CSP process in CLP syntax, which can be automated easily by syntax rewriting. A relation of the form *proc*(N, P) is used to present a process P with name N. For instance, Figure 3.1 is the syntax encoding of process TVM (Example 2.2.1) in CLP, which is a recursive process with name *tvm*.

[1] 2 means predicate "sequential" has two parameters.

Timed CSP Process	CLP presentation
SKIP	skip
STOP	stop
RUN	run
a $\rightarrow$ P	eventprefix(a, P)
a $\stackrel{t}{\rightarrow}$ P	delay(a, t, p)
P1 $\Box$ P2	extchoice(P1, P2)
P1 $\sqcap$ P2	intchoice(P1, P2)
P1 $_X\|\|_Y$ P2	parallel(P1, P2, X, Y)
P1 $\|[\, X \,]\|$ P2	parallel(P1, P2, X)
P1 $\|\|\|$ P2	interleave(P1, P2)
P1; P2	sequential(P1, P2)
P1 $\triangledown$ P2	interrupt(P1, P2)
P1 $\stackrel{d}{\triangleright}$ P2	timeout(P1, P2, d)
WAIT d	wait(d)
P1 $\triangledown_d$ P2	tinterrupt(P1, P2, d)
μ X @ P(X)	recursion(P)

Table 3.1: Library of Timed CSP processes in CLP

3.1.2 Laws and Simplification

[89] introduces the concept of process equivalence $P_1 = P_2$ when $P_1 =_T P_2$, $P_1 =_{SF} P_2$ and $P_1 = FDI\, P_2$ hold.

A set of laws are defined associated with each operator, where some of them can be used for process simplification. For example, $\Box$-idem law $P \Box P = P$, which states that offering a choice between two copies of the same process is identical to the process itself, can simplify process $P \Box P$ to P. A set of such laws are selected and implemented in the reasoning engine to fulfil this purpose, which are shown below.

$proc(c1, delay(coffee, eventprefix(dispatchcoffee, tvm), 3)).$
$proc(c2, delay(tea, eventprefix(dispatchtea, tvm), 2)).$
$proc(c3, eventprefixc(reqrelease, delay(release, tvm, 2))).$
$proc(choices, extchoice(extchoice(C, T), R))$
$\quad :- proc(c1, C), proc(c2, T), proc(c3, R).$
$proc(to, timeout(C, eventprefix(release, tvm), 10))$
$\quad :- proc(choices, C).$
$proc(tvm, eventprefix(coin, tvm)))$
$\quad :- proc(to, P).$

Figure 3.1: Timed Vending Machine in CLP

$P \,\Box\, P = P$	$\langle \Box\text{-}idem \rangle$
$P_1 \,\Box\, P_2 = P_2 \,\Box\, P_1$	$\langle \Box\text{-}sym \rangle$
$P \,\Box\, \textsc{Stop} = P$	$\langle \Box\text{-}unit \rangle$
$P \sqcap P = P$	$\langle \sqcap\text{-}idem \rangle$
$P_1 \sqcap P_2 = P_2 \sqcap P_1$	$\langle \sqcap\text{-}sym \rangle$
$\textsc{Skip} \;{}_A\|\|_B\; \textsc{Skip} = \textsc{Skip}$	$\langle \|\|\text{-}term1 \rangle$
$P \;{}_A\|\|_A\; P = P \qquad if \alpha(P) \subseteq A$	$\langle \|\|\text{-}idem \rangle$
$\textsc{Skip} \;\|\|\|\; \textsc{Skip} = \textsc{Skip}$	$\langle \|\|\|\text{-}term1 \rangle$
$P \;\|\|\|\; \textsc{Skip} = P$	$\langle \|\|\|\text{-}unit \rangle$
$P_1 \;\|\|\|\; P_2 = P_2 \;\|\|\|\; P_1$	$\langle \|\|\|\text{-}sym \rangle$
$\textsc{Skip};\; P = P$	$\langle ;\text{-}unit\text{-}I \rangle$
$P;\; \textsc{Skip} =_T P$	$\langle ;\text{-}unit\text{-}r_T \rangle$
$\textsc{Stop};\; P = \textsc{Stop}$	$\langle ;\text{-}zero\text{-}I \rangle$
$\textsc{Stop} \bigtriangledown P = P$	$\langle \bigtriangledown\text{-}unit\text{-}I \rangle$
$P \bigtriangledown \textsc{Stop} = P$	$\langle \bigtriangledown\text{-}unit\text{-}r \rangle$
$\textsc{Skip} \bigtriangledown P = \textsc{Skip} \,\Box\, P$	$\langle \bigtriangledown\text{-}term \rangle$

Laws for timed operators timeout $P_1 \overset{d}{\triangleright} P_2$, delay $a \overset{d}{\rightarrow} P$, wait WAIT and timed interrupt

$P_1 \bigtriangledown_d P_2$ as shown as follows.

$$\begin{array}{ll}
\textsc{Stop} \overset{d}{\triangleright} Q_1 = \textsc{Wait}\ d;\ Q_1 & \langle \triangleright\text{-}delay \rangle \\
Q_1 \overset{d}{\triangleright} (Q_2 \overset{d'}{\triangleright} Q_3) = (Q_1 \overset{d}{\triangleright} Q_2) \overset{d+d'}{\triangleright} Q_3 & \langle \triangleright\text{-}assoc \rangle \\
(\textsc{Wait}\ d;\ Q_1) \overset{d+d'}{\triangleright} Q_2) = \textsc{Wait}\ d;\ (Q_1 \overset{d'}{\triangleright} Q_2) & \langle \triangleright\text{-}delay\text{-}1 \rangle \\
(\textsc{Wait}(d+d');\ Q_1) \overset{d}{\triangleright} Q_2) = \textsc{Wait}\ d;\ Q_2 & \langle \triangleright\text{-}delay\text{-}2 \rangle \\
\textsc{Wait}\ d;\ \textsc{Wait}\ d' = \textsc{Wait}(d+d') & \langle delay\text{-}sum \rangle \\
a \overset{d}{\rightarrow} P = a \rightarrow \textsc{Wait}\ d;\ P & \langle delay\text{-}prefix \rangle \\
(\textsc{Wait}\ d;\ Q_1)\ {}_A||_B\ (\textsc{Wait}\ d;\ Q_2) = \textsc{Wait}\ d;\ (Q_1\ {}_A||_B\ Q_2) & \langle delay\text{-}||\text{-}dist \rangle \\
(\textsc{Wait}\ d;\ Q_1)\ |||\ (\textsc{Wait}\ d;\ Q_2) = \textsc{Wait}\ d;\ (Q_1\ |||\ Q_2) & \langle delay\text{-}|||\text{-}dist \rangle \\
(\textsc{Wait}\ d;\ Q_1)\,|[\,A\,]|\,(\textsc{Wait}\ d;\ Q_2) = \textsc{Wait}\ d;\ (Q_1\,|[\,A\,]|\,Q_2) & \langle delay\text{-}|[\,A\,]|\text{-}dist \rangle \\
Q_1 \bigtriangledown_d (Q_2 \bigtriangledown_{d'} Q_3) = (Q_1 \bigtriangledown_d Q_2) \bigtriangledown_{d+d'} Q_3 & \langle \bigtriangledown_d\text{-}assoc \rangle \\
\textsc{Stop} \bigtriangledown_d P = \textsc{Wait}\ d;\ Q_1 & \langle stop\text{-}\bigtriangledown_d\text{-}delay \rangle \\
(\textsc{Wait}\ d;\ Q_1) \bigtriangledown_{d+d'} Q_2) = \textsc{Wait}\ d;\ (Q_1 \bigtriangledown_{d'} Q_2) & \langle \bigtriangledown\text{-}delay\text{-}1 \rangle \\
(\textsc{Wait}(d+d');\ Q_1) \bigtriangledown_d Q_2) = \textsc{Wait}\ d;\ Q_2 & \langle \bigtriangledown\text{-}delay\text{-}2 \rangle
\end{array}$$

For the purpose of simplification, we encode the laws described above into CLP clauses in a structured way. A CLP relation $law(P, Q)$ is introduced to find the simplified version of process Q of P. $law/2$ rules for each operator are defined recursively which is illustrated as

follows.

$law(extchoice(P, P), P) : -!.$
$law(extchoice(P, stop), P) : -!.$
$law(extchoice(stop, P), P) : -!.$
$law(intchoice(P, P), P) : -!.$
$law(parallel(skip, skip, \_), skip) : -!.$
$law(parallel(P, P, A, B), P) : -setequal(A, B), !.$
$law(parallel(skip, skip, \_), skip) : -!.$
$law(parallel(P, P, \_), P) : -!.$
$law(interleave(P, stop), P) : -!.$
$law(interleave(stop, P), P) : -!.$
$law(interleave(P, skip), P) : -!.$
$law(interleave(skip, P), P) : -!.$
$law(sequential(skip, P), P) : -!.$
$law(sequential(P, skip), P) : -!.$
$law(sequential(stop, \_), stop) : -!.$
$law(interrupt(stop, P), P) : -!.$
$law(interrupt(P, stop), P) : -!.$
$law(extchoice(P, Q), S) : -not(P = stop;\ Q = stop, P = Q),$
$law(P, P1), law(Q, Q1), (P1 = P, Q1 = Q) - >$
$S = extchoice(P, Q);\ law(extchoice(P1, Q1), S), !.$
$law(intchoice(P, Q), S) : -not(P = Q), law(P, P1), law(Q, Q1),$
$(P1 = P, Q1 = Q) - > S = intchoice(P, Q);\ law(intchoice(P1, Q1), S).$
$law(parallel(P, Q, A, B), parallel(P, Q, A, B)) : -$
$not(P = skip, setequal(A, B)), !.$
$law(parallel(P, Q, A, B), P) : -setequal(A, B), not(P = Q),$
$law(P, P1), law(Q, Q1), (P1 = P, Q1 = Q)$
$- > S = parallel(P, Q, A, B);\ law(parallel(P1, Q1, A, A), S).$
$law(parallel(P, Q, A), S) : -not(P = Q), law(P, P1), law(Q, Q1),$
$(P1 = P, Q1 = Q) - > S = parallel(P, Q, A);\ law(parallel(P1, Q1, \_), S).$
$law(interleave(P, Q), S) : -not(P = skip;\ Q = skip),$
$law(P, P1), law(Q, Q1), (P1 = P, Q1 = Q)$
$- > S = interleave(P, Q);\ law(interleave(P1, Q1), S).$
$law(sequential(P, Q), S) : -not(P = skip;\ P = stop;\ Q = skip),$
$law(P, P1), law(Q, Q1), (P1 = P, Q1 = Q)$
$- > S = sequential(P, Q);\ law(sequential(P1, Q1), S).$
$law(interrupt(skip, P), extchoice(skip, P)) : -!.$
$law(interrupt(P, Q), S) : -not(P = skip;\ P = stop;\ Q = stop),$
$law(P, P1), law(Q, Q1), (P1 = P, Q1 = Q)$
$- > S = interrupt(P, Q);\ law(interrupt(P1, Q1), S).$

3.2 Modeling Operational Semantics of Timed CSP in CLP

Operational semantics provides a way of interpreting a language - of stepping through executions of programs written in that language. It therefore offers a direct intuition of how program constructs are intended to behave, in contrast with denotational semantics which often abstract away from such considerations, and with algebraic approaches, where operators are defined in terms of the algebraic laws when satisfy.

This section is devoted to an encoding of the semantics of Timed CSP in CLP. The practical implication is that we may then use powerful constraint solver like CLP($\mathcal{R}$) [55] to do various proving over systems modelled using Timed CSP. Both the operational semantics and denotational semantics are encoded. The encoding of operational semantics serves most of our purposes.

The operational semantics of Timed CSP is precisely defined by Schneider [90] using two relations: an evolution relation and a timed event transition relation. It is straightforward to verify that our encoding conforms the two relations in [90].

A relation of the form *tos(P1,T1,E,P2,T2)* is used to denote the *t*imed *o*perational *s*emantics, by capturing both evolution relations and timed event transition relations. Predicate *tos(P1,T1,E,P2,T2)* is true if the process *P1* may evolve to *P2* through either a timed transition, i.e., let *T2-T1* time units pass, or an event transition by engaging an abstract event instantly[2]. The relation *tos* defines a transition system interpretation of a Timed CSP process, where the state is identified by the combination of the process expression and the time variable. Using tabling mechanism offered in some of the constraint solvers like CLP($\mathcal{R}$) [55] or XSB [105], the termination of the derivation sequence based on relation *tos* depends on the finiteness of the reachable process expressions from the initial one. Therefore, if a process is irregular, proving of goals which need to explore all reachable process expressions is not feasible.

[2]Or both at the same time by engaging an nontrivial action which takes time (necessary for only extensions to Timed CSP like TCOZ [74] where E could be a complicated computation)

However, even for irregular processes, interesting proving like existence of a trace is still possible.

We define the *tos* relation in terms of each and every operator of Timed CSP. For the moment, we assume the process is not parameterized and we shall handle parameterized processes uniformly in Section 3.3.1.

3.2.1 Primitive process

Stop, Skipand Run

The operational semantics of the primitive process expressions in Timed CSP are defined through the following transitions:

$$\frac{}{\textsc{Stop} \overset{d}{\rightsquigarrow} \textsc{Stop}}$$

$$\frac{}{\textsc{Skip} \rightarrow \checkmark\textsc{Stop}}$$

$$\frac{}{\textsc{Skip} \overset{d}{\rightsquigarrow} \textsc{Skip}}$$

The rules associated with the Skip, Stop and Run processes are as the following:

$tos(stop, T1, [], stop, T2) : -D >= 0, T2 = T1 + D.$
$tos(skip, T, [termination], stop, T).$
$tos(skip, T1, [], skip, T1 + D) : -D >= 0.$
$tos(run, T, [\_], run, T).$
$tos(run, T1, [], run, T2) : -D >= 0, T2 = T1 + D.$

The only transition for process Stop is time elapsing. Process Skip may choose to wait some time before engaging event *termination* which is our choice of representation for event ✓ in CLP. Process Run may either let time pass or engage any event.

Event prefix

Event prefix expression $a \rightarrow P$ describes a process which is prepared to engage in the event a after which it will behave as P. The semantics of this event prefix expression is illustrated in the following two rules:

$$\frac{}{(a \rightarrow P) \xrightarrow{a} P}$$

$$\frac{}{(a \rightarrow P) \overset{d}{\rightsquigarrow} (a \rightarrow P)}$$

The $\rightarrow$ represents an event transition, whereas $\rightsquigarrow$ represents an evolution transition. The rules associated with the event prefix operator are as the following:

$tos(eventprefix(E, P), T, [E], P, T).$
$tos(eventprefix(E, P), T1, [], eventprefix(E, P), T1 + D) : -D > 0.$

The first rule states that this process is initially able to perform only event a, and after performing a it behaves as P. The second rule states the process remains the same, but allows d time unites elapse without any event being engaged.

3.2.2 Choice

Timeout

The timeout operator $P_1 \overset{d}{\triangleright} P_2$ introduces a way of describing time-sensitive process behavior, which offers a time-sensitive choice between $P1$ and $P2$. The operational semantics is defined using the following four inference rules:

$$\frac{P_1 \xrightarrow{a} P_1'}{P_1 \overset{d}{\triangleright} P_2 \xrightarrow{a} P_2}$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{P_1 \overset{d}{\triangleright} P_2 \xrightarrow{\tau} P_1' \overset{d}{\triangleright} P_2}$$

$$\frac{P_1 \overset{d'}{\rightsquigarrow} P_1'}{P_1 \overset{d}{\triangleright} P_2 \overset{d'}{\rightsquigarrow} P_1' \overset{d-d'}{\triangleright} P_2} [\, 0 < d' \leqslant d \,]$$

$$\frac{}{P_1 \overset{0}{\triangleright} P_2 \overset{\tau}{\rightarrow} Q2}$$

Where the first d time unites of any execution will have transitions identical to an execution of P_1, since all transitions before the timeout occurs correspond to transitions of P_1. The corresponding CLP rules of timeout process is specified as follows:

$tos(timeout(Q1, \_, \_), T, [E], P, T) :- tos(Q1, T, [E], P, T).$
$tos(timeout(Q1, Q2, D), T, [tau], timeout(P, Q2, D), T) :- tos(Q1, T, [tau], P, T).$
$tos(timeout(Q1, Q2, D), T1, [], timeout(P, Q2, D - T), T1 + T)$
$\quad :- T > 0, T <= D, tos(Q1, T1, [], P, T1 + T).$
$tos(timeout(\_, Q2, 0), T, [tau], Q2, T).$

The first rules states if P_1 performs any external event (not τ event), the timeout choice is resolved in favor of P_1, and P_2 is discarded. The second rules states if P_1 performs any internal event (τ), it does not resolve the choice. The evolution transition relation is modeled in the third rule where the process remains timeout with d' (given 0 < d' ⩽ d) time unites elapse. The last rule states if P_1 has not perform any external event within its allotted time, which means d reduced to 0, the choice is resolved in favor of P_2.

Delay

The delay event prefix (delay process) $a \overset{d}{\rightarrow} P$ delays process P by d time units after engaging event a. In [90], a delay is defined as a process immediately following the occurrence of an event occurs frequently enough in process description to warrant its own abbreviated forms:

$$a \overset{d}{\rightarrow} P = a \rightarrow a \rightarrow (\textsc{Stop} \overset{d}{\triangleright} P)$$

We model the semantics of $a \overset{d}{\rightarrow} P$ the same as $a \rightarrow a \rightarrow (\textsc{Stop} \overset{d}{\triangleright} P)$, where corresponding $tos/5$ rule is defined as:

$tos(delay(A, Q, D), T1, E, P, T2)$
$\quad :- tos(eventprefix(A, timeout(stop, Q, D)), T1, E, P, T2).$

Wait

The WAIT d process, which is the initial primitive Timed CSP added, delays for exactly d time unites before terminating. It is defined using a timeout construction [90]:

$$\text{WAIT } d = \text{STOP} \overset{d}{\triangleright} \text{SKIP}$$

The process WAIT d can perform no events for the first d time units of its execution, but after that delay it terminates and its subsequent behavior is that of SKIP. The CLP relation for WAIT d $tos/5$ is defined exactly as the STOP $\overset{d}{\triangleright}$ SKIP:

$$tos(wait(D), T1, E, P, T2) \\ \quad :- tos(timeout(stop, skip, D), T1, E, P, T2).$$

External Choice

Choice operators are very important in Timed CSP. The external choice $P_1 \,\Box\, P_2$ offers a choice between processes P_1 and P_2, which is resolved at the instant the first visible event occurs, in favor of the process which performs it. The rules for deriving transitions of a choice are given as follows:

$$\frac{P_1 \xrightarrow{a} P_1'}{\begin{array}{l} P_1 \,\Box\, P_2 \xrightarrow{a} P_1' \\ P_2 \,\Box\, P_1 \xrightarrow{a} P_1' \end{array}}$$

$$\frac{P_1 \xrightarrow{\tau} P_1'}{\begin{array}{l} P_1 \,\Box\, P_2 \xrightarrow{\tau} P_1' \,\Box\, P_2 \\ P_2 \,\Box\, P_1 \xrightarrow{\tau} P_2 \,\Box\, P_1' \end{array}}$$

$$\frac{P_1 \overset{d}{\rightsquigarrow} P_1' \quad P_2 \overset{d}{\rightsquigarrow} P_2'}{P_1 \,\Box\, P_2 \overset{d}{\rightsquigarrow} P_1' \,\Box\, P_2'}$$

The operational semantics of process $P_1 \,\square\, P_2$ is defined as five $tos/5$ rules:

$$
\begin{aligned}
&tos(extchoice(P1, \_), T, [E], P3, T) : -tos(P1, T, [E], P3, T).\\
&tos(extchoice(\_, P2), T, [E], P4, T) : -tos(P2, T, [E], P4, T).\\
&tos(extchoice(P1, P2), T, [tau], extchoice(P3, P2), T)\\
&\quad : -tos(P1, T, [tau], P3, T).\\
&tos(extchoice(P1, P2), T, [tau], extchoice(P1, P4), T)\\
&\quad : -tos(P2, T, [tau], P4, T).\\
&tos(extchoice(P1, P2), T1, [], extchoice(P3, P4), T2)\\
&\quad : -T2 > T1, tos(P1, T1, [], P3, T2), tos(P2, T1, [], P4, T2).
\end{aligned}
$$

Where the first two rules captures the first deriving rules. The second deriving rules are modeled in the next two $tos/5$ rules and the last $tos/5$ states the evolution transition.

3.2.3 Concurrency

Alphabetized parallel

In the operational semantics, the event transition and evolution transition associated with the alphabetized parallel composition operator the alphabetized parallel composition operator $P_1 \;{}_X||_Y\; P_2$ are illustrated as the following [90]:

$$\frac{P_1 \xrightarrow{e} P_1'}{P_1 \;{}_X||_Y\; P_2 \xrightarrow{e} P_1' \;{}_X||_Y\; P_2} [\; e \in X \cap \{\tau\} \setminus Y \;]$$

$$\frac{P_2 \xrightarrow{e} P_2'}{P_1 \;{}_X||_Y\; P_2 \xrightarrow{e} P_1 \;{}_X||_Y\; P_2'} [\; e \in Y \cap \{\tau\} \setminus X \;]$$

$$\frac{P_1 \xrightarrow{e} P_1', P_2 \xrightarrow{e} P_2'}{P_1 \;{}_X||_Y\; P_2 \xrightarrow{e} P_1' \;{}_X||_Y\; P_2'} [\; e \in X \cap Y \;]$$

$$\frac{P_1 \overset{d}{\rightsquigarrow} P_1', P_2 \overset{d}{\rightsquigarrow} P_2'}{P_1 \;{}_X||_Y\; P_2 \overset{d}{\rightsquigarrow} P_1' \;{}_X||_Y\; P_2'}$$

The $\rightarrow$ represents an event transition, whereas $\rightsquigarrow$ represents an evolution transition. The rules associated with the alphabetized parallel composition operator are as the following:

$$
\begin{aligned}
&tos(para(P1, P2, X, Y), T, [E], para(P3, P2, X, Y), T)\\
&\quad :- tos(P1, T, [E], P3, T), member(E, X), not_member(E, Y).\\
&tos(para(P1, P2, X, Y), T, [E], para(P1, P4, X, Y), T)\\
&\quad :- os(P2, T, [E], P4, T), member(E, Y), not_member(E, X).\\
&tos(para(P1, P2, X, Y), T, [E], para(P3, P4, X, Y), T)\\
&\quad :- tos(P1, T, E, P3, T), tos(P2, T, E, P4, T),\\
&\qquad member(E, X), member(E, Y).\\
&tos(para(P1, P2, X, Y), T1, [], para(P3, P4, X, Y), T1 + D)\\
&\quad :- tos(P1, T1, [], P3, T1 + D), tos(P2, T1, [], P4, T1 + D).
\end{aligned}
$$

clp relation $para(P1, P2, X, Y)$ is to present a parallel process $P_1 \ {}_X||_Y \ P_2$. The first two rules state that either of the components may engage an event as long as the event is not shared. The third rule states that a shared event can only be engaged simultaneously by both components. The last expresses states that the composition may allow time elapsing as long as both the components do. Other parallel composition operation, like $|[\, X \,]|$ and $|||$, can be defined as special cases of the alphabetized parallel composition operator straightforwardly.

interleaving

Concurrency execution of two processes without synchronization is described by an interleaving process, $P_1 \ ||| \ P_2$. The introduction of time to the interleaving operator is entirely similar to the approach taken for the parallel composition. The operational semantics rules for deriving the transitions for an interleaved combination are as follows:

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \ ||| \ P_2 \xrightarrow{\mu} P_1' \ ||| \ P_2} [\, \mu \neq \checkmark \,]$$

$$\frac{P_2 \xrightarrow{\mu} P_2'}{P_2 \ ||| \ P_1 \xrightarrow{\mu} P_2' \ ||| \ P_1} [\, \mu \neq \checkmark \,]$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1' \qquad P_2 \xrightarrow{\checkmark} P_2'}{P_1 \ ||| \ P_2 \xrightarrow{\checkmark} P_1' \ ||| \ P_2'}$$

$$\frac{P_1 \overset{d}{\rightsquigarrow} P_1' \qquad P_2 \overset{d}{\rightsquigarrow} P_2'}{P_1 \;|||\; P_2 \overset{d}{\rightsquigarrow} P_1' \;|||\; P_2'}$$

The rules associated with the interleaving composition operator are as the following:

$$\begin{aligned}
&tos(interleave(P1, P2), T, [E], interleave(P3, P2), T)\\
&\quad :- tos(P1, T, [E], P3, T).\\
&tos(interleave(P1, P2), T, [E], interleave(P1, P3), T)\\
&\quad :- tos(P2, T, [E], P3, T).\\
&tos(interleave(P1, P2), T, [termination], interleave(P3, P4), T)\\
&\quad :- tos(P1, T, [termination], P3, T), tos(P2, T, [termination], P4, T).\\
&tos(interleave(P1, P2), T1, [], interleave(P3, P4), T1 + D)\\
&\quad :- D > 0, tos(P1, T1, [], P3, T1 + D), tos(P2, T1, [], P4, T1 + D).
\end{aligned}$$

The clp relation $interleaving(P1, P2)$ is used to present the interleaving process $P_1 \;|||\; P_2$. The First two rules state that either of the component can perform an external event independently, which will be appended to the trace of this process. The fifth rule models the third transition rule of the operational semantics, which states that both components must engage the termination event ✓ simultaneously. The last expresses states that the composition may allow time elapsing as long as both the components do.

3.2.4 Flow of control

Sequential

The mechanism for transferring control from a terminated process to another process is sequential composition: $P_1;\; P_2$. It allows control to pass to a second process when the first one terminates successfully, as indicated by the occurrence of the termination event ✓. The operational semantics rules for deriving the transitions for a sequential combination are as follows:

$$\frac{P_1 \overset{\mu}{\rightarrow} P_1'}{P_1;\; P_2 \overset{\mu}{\rightarrow} P_1';\; P_2}\; [\,\mu \neq \checkmark\,]$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1;\ P_2 \xrightarrow{\tau} P_2}$$

$$\frac{P_1 \overset{d}{\rightsquigarrow} P_1' \qquad P_2 \overset{d}{\rightsquigarrow} P_2'}{P_1 \mid\mid\mid P_2 \overset{d}{\rightsquigarrow} P_1' \mid\mid\mid P_2'}$$

The rules associated with the sequential composition operator are as the following:

$$\begin{aligned}
&tos(sequential(P1, P2),\ T, [E], sequential(P3, P2),\ T)\\
&\quad :- tos(P1,\ T, [E], P3,\ T), not(E = termination).\\
&tos(sequential(P1, P2),\ T, [tau], P2,\ T)\\
&\quad :- tos(P1,\ T, [termination],,\ T).\\
&tos(sequential(P1, P2),\ T1, [], sequential(P3, P2),\ T1 + D)\\
&\quad :- D > 0, tos(P1,\ T1, [], P3,\ T1 + D), not(tos(P1, _, [termination], _, _)).
\end{aligned}$$

CLP relation $sequentail(P1, P2)$ is defined to denote the sequential process $P_1;\ P_2$. The first operational semantics rule is translated into the first tos/5 rule of sequential/3, which states that the sequential composition $P_1;\ P_2$ initially executes as P_1 before P_1 terminates. The second rule states that when P_1 terminates, its ✓ event becomes internal to the composition and it passes control to process P_2. The last expresses states that the composition may allow time elapsing as long as P_1 does, only if P_1's termination event is not enabled. The last rule constraints that the passing of control is urgent.

Interrupt

The interrupt construction $P_1 \triangledown P_2$ allows the first process P_1 to execute, but it may be interrupted at any time by an event from process P_2. The process P_2 is also able to perform internal events without triggering the interrupt, which means that external interrupt events maybe offered and retracted by P_2 as the execution unfolds. The operational semantics rules for deriving the transitions for an interrupt combination are as follows:

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \triangledown P_2 \xrightarrow{\mu} P_1' \triangledown P_2}\ [\ \mu \neq \checkmark\ ]$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1 \triangledown P_2 \xrightarrow{\checkmark} P_1'}$$

$$\frac{P_2 \xrightarrow{\tau} P_2'}{P_1 \triangledown P_2 \xrightarrow{\mu} P_1 \triangledown P_2'}$$

$$\frac{P_2 \xrightarrow{a} P_2'}{P_1 \triangledown P_2 \xrightarrow{a} P_2}$$

$$\frac{P_1 \overset{d}{\rightsquigarrow} P_1' \qquad P_2 \overset{d}{\rightsquigarrow} P_2'}{P_1 \triangledown P_2 \overset{d}{\rightsquigarrow} P_1' \triangledown P_2'}$$

The rules associated with the interrupt composition operator are as the following:

$tos(interrupt(P1, P2), T, [E], interrupt(P3, P2), T)$
$\quad :- tos(P1, T, [E], P3, T), not(E == tau).$
$tos(interrupt(P1, P2), T, [], interrupt(P3, P2), T)$
$\quad :- tos(P1, T, [tau], P3, T).$
$tos(interrupt(P1, P2), T, [termination], interrupt(P3, P2), T)$
$\quad :- tos(P1, T, [termination], P3, T).$
$tos(interrupt(P1, P2), T, [], interrupt(P1, P3), T)$
$\quad :- tos(P1, T, [tau], P3, T).$
$tos(interrupt(\_, P2), T, [E], P3, T)$
$\quad :- tos(P2, T, [E], P3, T), not(E = tau).$
$tos(interrupt(P1, P2), T1, [], interrupt(P3, P4), T1 + D)$
$\quad :- D > 0, tos(P1, T1, [], P3, T1 + D), tos(P2, T1, [], P4, T1 + D).$

CLP relation $interrupt(P1, P2)$ is defined to denote the interrupt process $P_1 \triangledown P_2$. The two expressions capture the first operational semantics rule, which states that the process is able to perform any execution of $P1$. The first expression captures the situation that P_1 performing an external event E which will be appended to the trace, while the second expression captures an internal event engagement which will not be added to the end of the trace. The third expression states that if P_1 terminate while it is executing then the entire structure is terminated and P_2 is discarded. The forth expression captures the third semantics rule when P_2 performs an internal event and the forth rule captures the moment when the interruption occurs where P_2 is executed to P_3 by engaging an external event E. The last expression illustrates the evolution transition.

Timed interrupt

The introduction of time allows an alternative approach to the interruption of processes. If the process is permitted to run for no more than a particular length of time, then the passage of time can itself trigger an interrupt to remove control from a process. The operational semantics rules for deriving the transitions for a timed interrupt combination $P_1 \triangledown_d P_2$. are as follows:

$$\frac{P_1 \xrightarrow{\mu} P_1'}{P_1 \triangledown_d P_2 \xrightarrow{\mu} P_1' \triangledown_d P_2} [\ \mu \neq \checkmark\ ]$$

$$\frac{P_1 \xrightarrow{\checkmark} P_1'}{P_1 \triangledown_d P_2 \xrightarrow{\checkmark} P_1'}$$

$$\frac{}{P_1 \triangledown_0 P_2 \xrightarrow{\tau} P_2}$$

$$\frac{P_1 \overset{d'}{\rightsquigarrow} P_1'}{P_1 \triangledown_d P_2 \overset{d'}{\rightsquigarrow} P_1' \triangledown_{d-d'} P_2'} [\ d' \leqslant d\ ]$$

The rules associated with the interrupt composition operator are as the following:

$tos(tinterrupt(\_, Q2, 0), T, [tau], Q2, T).$
$tos(tinterrupt(Q1, Q2, D), T, [E], tinterrupt(Q3, Q2, D), T)$
$\quad :- tos(Q1, T, [E], Q3, T), not(E == termination).$
$tos(tinterrupt(Q1, \_, \_), T, [termination], Q3, T)$
$\quad :- tos(Q1, T, [termination], Q3, T).$
$tos(tinterrupt(Q1, Q2, D), T1, [], tinterrupt(Q3, Q2, D - T), T1 + T)$
$\quad :- not(D == 0), T > 0, T <= D, tos(Q1, T1, [t(T)], Q3, T1 + T).$

A CLP relation $tinterrupt(P1, P2, D)$ is defined to denote the timed interrupt process $P_1 \triangledown_d P_2$.

There is a clear one-to-one correspondence between our rules and the operators which are fully illustrated in Appendix A. Therefore, the soundness of the encoding can be proved by showing there is a bi-simulation relationship between the transition system interpretation defined in [90] and ours, and the bi-simulation relationship can be proved easily via a structural induction.

3.3 Modeling Denotational Semantics of Timed CSP in CLP

In Section 3.2 operational semantics is encoded into CLP, where the encoding of operational semantics serves most of our purposes. Nevertheless the encoding of the denotational semantics offers an alternative way of proving systems modelled in Timed CSP as well as the correctness of the encoding itself.

In this section, we encode both the timed traces and the timed failures model of Timed CSP, where the semantics of a Timed CSP process is represented by a set of timed traces or a set of timed failures [89]. A timed failure is a record of an execution, consisting of a timed trace which contains information about event performed, and a timed refusal which contains information about when events could be refused. In contrast to the operational semantics, which focuses on a single step at once, the denotational semantics captures all possible observations of systems modelled using Timed CSP. Therefore, it is easier to prove over all possible behaviors in the denotational semantics model.

In the following, we illustrate our encoding using only a few fundamental constructors for the sake of space saving. A relation *timedfailure(P, f(Tr, R))* is defined to capture the timed failure semantics, where P is a process expression and Tr is a sequence of timed events and R is a set of timed refusals. For instance,

$$
\begin{array}{l}
timedfailure(stop, failure([], \_)).\\
timedfailure(skip, failure([], R))\\
\quad :- sigma(R, S), not_member(termination, S).\\
timedfailure(skip, failure([tevent(T, termination)], R))\\
\quad :- T >= 0, before(R, T, Z), sigma(Z, N),\\
\qquad not_member(termination, N).
\end{array}
$$

The relation *sigma(P, S)* is used to retrieve all events S in a process expression P, i.e., $S = \sigma(P)$. Similarly, the relation *before(R, T, Z)* is defined accordingly as $Z = R \upharpoonright T$, i.e., the refusals before time T. Basically, the first rule states that the failures of process STOP are an empty trace with all possible refusals. Process SKIP refuses everything until the occurrence of event *termination*, and all events are refused afterwards. As for compositional operators,

we take the interface parallel composition operator as an example.

$$
\begin{aligned}
&timedfailure(parallel(Q1, Q2, A), failure(S, N)) \\
&\quad :- timedfailure(Q1, failure(S1, N1)), \\
&\qquad timedfailure(Q2, failure(S2, N2)), union(N1, N2, N), \\
&\qquad union(A, [termination], AT), remove(N1, AT, N11), \\
&\qquad remove(N2, AT, N22), setequal(N11, N22), tsynch(S1, S2, A, S).
\end{aligned}
$$

The relation *union(X, Y, Z)* is the set union, i.e., $Z = X \cup Y$. The relation *remove(X, Y, Z)* is the set subtraction, i.e., $Z = X \setminus Y$. The relation *tsynch* defines the ways in which a trace tr_1 from component $Q1$ and a trace tr_2 from component $Q2$ can be combined to form a trace of the parallel (formal definition in [89]). The interface parallel operator requires synchronization on events from the interface event set A, and interleaving on events not in A.

Notice that the denotational semantics focuses on observations of the system, which allows us to query the system behaviors as a whole. For instance, it is more straightforward to check timewise refinement using the denotational semantics, and irregular processes can be handled if we replace the recursion using its fixed point. However, because there is no guarantee that the derivation sequence is terminating, we have to limit the height of the proving tree.

3.3.1 Handling Extensions to Timed CSP

Timed CSP is introduced in [82]. Since then, various extensions of Timed CSP have been proposed. In this work, we identify some of the effective extensions and show that they can be encoded in the CLP framework. For instance, the idea of *signal* by Davies [17] is a simple yet useful extension to capture liveness as well as model broadcasting effectively. The motivation of the concept *signal* is that when describing the behavior of a real-time process, we may wish to include instantaneous observable events that are not synchronization. For example, an audible bell might form part of the user interface to a telephone network, even though the bell may ring (a *signal*) without the cooperation of the user. Informally, *signal*

events are distinguished events that will occur as soon as they become available, and will propagate through parallel composition. A process may ignore any signal performed by another process, unless it is waiting to perform the corresponding synchronization. For any observation that can be extended into the future, the only events that must be observed are signals. Therefore, signals are useful both for modelling broadcast communication and specifying liveness conditions, i.e., some events must be engaged.

$$
\begin{array}{l}
sigTF(eventprefix(E,\_,\_), sigfailure([], X, T)) \\
\quad :- not(E == sig(\_)), sigma(X, Z), \\
\qquad not_member(E, Z), end(X, T1), T >= T1. \\
sigTF(eventprefix(E, P, D), sigfailure([tevent(T, E) \mid XS], Y, T1 + D + T)) \\
\quad :- T >= 0, not(E == sig(\_)), \\
\qquad sigTF(P, sigfailure(S, Y1), T1), \\
\qquad backthrough(Y, T + D, Y1), begin(S, T2), \\
\qquad T2 >= T + D, end(S, Y, T3), max(T, T3, T4), \\
\qquad T1 + D + T >= T4, before(Y, T, Z), sigma(Z, N), \\
\qquad not_member(E, N), delay(S, T + D, XS). \\
sigTF(eventprefix(sig(E), P, D), sigfailure([], [], 0)). \\
sigTF(eventprefix(sig(E), P, D), sigfailure([tevent(0, E) \mid XS], Y, T)) \\
\quad :- sigTF(P, failure(S, Y1), T1), \\
\qquad backthrough(Y, T + D, Y1), T = T1 + D, \\
\qquad before(Y, T, Z), sigma(Z, N), \\
\qquad not_member(E, N), delay(S, T + D, XS).
\end{array}
$$

The relation $sigTF(P, sigfailure(Tt, Tr, T))$ is used to capture this time failure semantics for signals, where P denotes the process, Tt is the timed trace, Tr denotes the timed refusal set and T denotes a time value. The CLP clauses illustrate the possible evolution of signal event prefixing. The first two clauses denote the semantics for event prefix process $a \rightarrow P$ where a is not a signal, while the last two denote the one with signal event $\hat{a}$, presented as $sig(a)$. In the above rules, *end(X,T)* computes the least upper bound of the time refusal X. *backthrough(Y,T,Y1)* represents the relation: Y - T = Y1, i.e., timed refusal $Y1$ is generated from Y by translating it backwards through time T. *begin(S, T)* retrieves the time of occurrence of the first event in timed trace S.

Another extension of special interest is Timed CSP integrated with state-based languages like Z [107] to model systems with not only complicated control flow but also complex data

structures [74, 92]. Instead of adopting a heavy language like TCOZ (Timed Communicating Object-Z [74]), we allow a finite number of variables to be associated with a process[3], called state variables. In addition, we allow a state update transition, i.e., instead of engaging an abstract event, the system may perform a state update which changes the valuation of the state variables. A state update is specified as a predicate involving state variables before and after the update, as in Z style where the after-variables are primed [107].

For instance, there is a fragment of the specification of this vending machine, in which we allow different coins to be inserted via a channel communication $coin?x$ where x is 10, 20 or 50, a data variable $Quota$ is requested to accumulate the amount of all coins inserted by the user.

$$Insert(Quota) \mathrel{\widehat{=}} coin?x \rightarrow AddQuota$$

where $AddQuota$ is an operation defined in Z, which is:

$$AddQuota \mathrel{\widehat{=}} [x?, quota, quota' : \mathbb{N} \mid quota' = quota + x?]$$

This Timed CSP specification corresponds to the following CLP clauses where both the pre and post values of the process parameter are presented as the parameters, namely $Quota1$ and $Quota2$, of the relation *proc*. The user is responsible to specify exactly how an action updates the data variables, e.g., adding the amount of the coin to $Quota$.

$$\begin{array}{l} proc(coin, eventpreifx(coin(X1), addquota), Quota1, Quota2) \\ \quad : -action(addquota, X1, Quota1, Quota2). \\ action(addquota, X1, Quota1, Quota2) : -Quota2 = Quota1 + X1. \end{array}$$

3.4 Verification of Timed CSP

This section is devoted to various proofs we may perform over systems modeled using Timed CSP and then encoded in CLP. These kind of assertions can be proved or disproved against

[3] which are of types supported by current tools for CLP.

a given real-time system. We also develope a number of shortcuts for easy querying and proving.

In order to explore the full state space, we define the following[4]:

$$
\begin{aligned}
&\mathit{reachable}(P, P, [], T1, T1).\\
&\mathit{reachable}(P, Q, N, T1, T2)\\
&\quad :- \mathit{tos}(P, T1, [\mathit{tau}], P1, T3), \mathit{reachable}(P1, Q, N, T3, T2).\\
&\mathit{reachable}(P, Q, [(E, T1) \mid N], T1, T2)\\
&\quad :- \mathit{tos}(P, T1, [E], P1, T3), \mathit{not}(E = t(\_);\ E == \mathit{tau}),\\
&\qquad \mathit{reachable}(P1, Q, N, T3, T2).\\
&\mathit{reachable}(P, Q, N, T1, T2)\\
&\quad :- \mathit{tos}(P, T1, [], P1, T3), \mathit{reachable}(P1, Q, N, T3, T2).
\end{aligned}
$$

The relation *reachable(P, Q, N, T1, T2)* states that it is possible to reach the process expression Q at time *T2* from P at time T1, with trace N.

reachable/5 is defined in four rules where the first states any process can remain at itself. The second rule states process P at $T1$ would reach process Q with trace N by firstly performing an internal event to process $P1$ at time $T3$ hence $P1$ would reach process Q with trace N. The third rule models the case P at $T1$ will reach Q with trace $[(E, T1) \mid N]$ by first performing an external event E at time $T1$ to process $P1$; then $P1$ is able to reach Q with trace N. The last rule captures the evolution transition from process P to $P1$ without appending any event to trace N.

3.4.1 Safety and Liveness Properties

Specifications are the properties that the design must satisfy. There are different ways to express properties. In state-based formalisms, properties are generally stated using temporal logics to specify how the system evolves over time. These properties are divided into two categories: safety properties and liveness properties.

We follow the informal classification of safety and liveness properties suggested by Lamport [61, 17]: a *safety property* is a requirement that 'nothing bad happens', while a *liveness*

[4]The possible state variables and local clocks are skipped for simplicity.

property insists that 'some good thing will eventually occur'. In either case, we must exclude undesirable behaviors from the semantic set of the process in question. In our model, a safety property corresponds to the requirement that a given event may not occur except under certain condition. For example:

- event a does not occur within time t after event b happens;
- if event a occurs, it must do so within time t after event b happens;
- event a may occur only at specific times.

[17] defines a safety specification on process Y in timed failure semantics as:

$$\forall\, s, X \bullet (s, X) \in Y \Rightarrow s \notin U$$

where U is a set of undesirable traces which violate the safety property. (s, X) is timed observation in the denotational semantics where s is a timed trace and X is a timed refusal. The formula says that if the safety specification is to be satisfiable, then there is no trace of Y be an undesired trace.

Using CLP, we may make explicit assertion which is neither just a safety assertion, nor just a liveness assertion. Yet it can be used for both purposes using a unique interpretation. An assertion is defined as Definition 3.

Definition 3 (Assertion) *An assertion is of the form $G \models G'$ where G is a nonterminal goal and G' is a possible terminal goal.*

An assertion can be used as a safety assertion by having G' as a terminal goal which should be the property/constraint that the system should satisfy. A safety property is generally expressed in the form:

$$G \models \Psi \text{ , } or \quad G, \neg\Psi \models \mathit{false}$$

which means there is no trace of the process violates the desired safety property Ψ.

Deadlock

One safety property of special interest is deadlock-freeness. The following clauses are used to prove it.

$$deadlock(P, T1) :- reachable(P, P1, N, T1, T2),$$
$$(not(tos(P1, T2, [], Q, T), tos(Q, T, [\_], \_, \_)); (tos(P1, T2, [], Q, \_);$$
$$not(tos, P1, T2, [\_], \_, \_))), printf("deadlock\ at : \%", [N]).$$

Basically, it states that a process P at time *T1* may result in deadlock if it can reach the process expression Q at time *T2* where no event transition is available neither at *T2* nor at any later moment. The last line outputs the deadlocked trace as a counterexample. Alternatively, we may present it as a result of the deadlock proving.

3.4.2 Trace-based Properties

We allow trace-based properties (safety or liveness[5]) that can be checked by exploring trace set partially. The retrieve of a timed trace is done by the predicate $trace(P, N, Q)$, which finds a sequence of events through which process expression P evolves to Q:

$$timed\_trace(P, N, Q) :- reachable(P, Q, N, 0, \_).$$

$$trace(P, N) :- timed\_trace(P, M, \_), remove\_time(M, N).$$

where $remove\_time(M, N)$ is a predefined CLP relation which removes times stamps of all event from a timed trace M.

We may prove that some event will always eventually be ready to be engaged using the following rule: where rule $member(N, E)$ returns true if event E appears at least once in the event sequence N, rule$not_member(N, E)$ returns true if event E is not an element of event sequence N.

$$finally(P, E) :- not_trace_W OMember(P, N, E).$$

[5]Liveness in CSP is a bit different from that in traditional temporal logic. In CSP, a process may wait forever before engaging an available event.

Predicate $\mathit{finally}(P, E)$ captures the idea that there is no such trace without event E in this process P. In other words, this process will eventually go to event E. Another property based on traces would be identifying the relationship among events, e.g., event A can never happen before (after) event B in a trace or trace fragment. Take the timed vending machine, which is defined in Example 2.2.1 for example, we would like to ensure that in a round of using the machine, the event *tea* will never be followed by an event *dispatchcoffee*.

Example 3.4.1 (Verification) For the timed vending machine (Example 2.2.1), we would like to check that it is deadlock-free by running the following goal and expecting failure:

$$? - \mathit{proc}(\mathit{vending}, P), \mathit{deadlock}(P, 0).$$

Moreover, we would expect that whenever we choose *tea*, it would never dispatch *coffee* instead of *tea*, which can be checked by the following goal:

$$? - \mathit{proc}(\mathit{vending}, P), \mathit{trace}(P, N), (\mathit{not}_m\mathit{ember}(N, \mathit{tea});\ \mathit{after}(N, \mathit{dispatchcoffee}, \mathit{tea})).$$

end

3.4.3 Time Related Checking

In addition to proving pre-specified assertions, one distinguished feature of our approach is that implicit assertions may be proved. In real time systems, sometimes it is very useful to find the lower and upper bound of a (time or data) variable such the applications like worst or best case analysis of execution time.

In our work, relation $\mathit{duration}(P)$ is defined to find the maximum execution time of a process P. If P is not terminating, value of *max* will be *infinity*. $\mathit{duration}(P)$ is defined as following rules:

$$\mathit{duration}(P) : -\mathit{duration\_time}(P, T, \_), \mathit{dump}([T]), \mathit{fail}.$$
$$\mathit{duration}(\_).$$
$$\mathit{duration}_t\mathit{ime}(P, T, N)$$
$$\quad : -\mathit{non\_termination}(P) - > (T = \mathit{infinity}, \mathit{fail});\ \mathit{reachable}(P, \mathit{stop}, N, 0, T).$$

where $non\_termination(P)$ is a predefined relation which is to check whether process P is terminating or not. If $non\_termination(P)$ returns true, T will be infinity and the rule checking is terminated.

We are also able to compute the duration of the engaging time of event e in some process P. Relation $engagedTime(P, E)$ is defined to fulfill this purpose.

$$engagedTime(P, E) :- happenTime(P, E, T, \_), dump([T]), fail.$$
$$engagedTime(\_, \_).$$
$$happenTime(P, E, T, X) :- reachable(P, \_, X, 0, T), last(X, [E \mid \_]).$$

Example 3.4.2 The process WAIT(2); $a \xrightarrow{3}$ SKIP should terminate in more than 5 time units, which can be identified by the following goal and expecting $T \geq 5$.

$$? - duration(sequential(wait(2), delay(a, skip, 3))).$$

After executing the goal, it would return result $T \in [5, infinity]$. The duration of engaging time of event a can be found by running the following goal and expecting $T \in [3, infinity]$.

$$? - engageTime(sequential(wait(2), delay(a, skip, 3)), a).$$

end

3.5 HORAE

Our engineering efforts have realized the proposed techniques in this chapter into a prototype **HORAE**, which is an interactive tool that provides composing and reasoning of Timed CSP process descriptions.

HORAE uses Constraint Logic Programming (CLP [54]) as underlying reasoning mechanism. CLP is designed for mechanized proving based on constraint solving. CLP has been successfully applied to model programs and transition systems for the purpose of verification. The main advantage of using CLP pertains to expressiveness. For example, [48] demonstrates the proof of some standard properties, as well as properties such as time

bounds between important events, on a CLP representation of Timed Safety Automata. [56] can even specify the property for testing whether a system of processes is symmetric, on a CLP representation of Timed Automata.

Built on the powerful constraint solver CLP($\mathcal{R}$) [55], **HORAE** can encode both operational and denotational semantics of Timed CSP processes in CLP($\mathcal{R}$) [29] and perform the verification of the Timed CSP models. The front-end of the tool is implemented in Java. The main features of this tool are listed as follows.

- Modeling Timed CSP models
- Specifying properties in a systematic way
- Verifying various kinds of properties with counterexamples provided if any, and
- Generating LaTeX presentation of the Timed CSP models.

An overview of **HORAE** is shown in Figure 3.2, which mainly consists of five components, i.e., a powerful GUI editor to compose Timed CSP models, an encoder which translates Timed CSP processes to CLP models, a property specifier which is used to specify properties in a systematic way, a verification engine which is used to verify properties and a LaTeX code generator. The software is implemented in Java and the CLP($\mathcal{R}$) is the CLP reasoning engine used inside.

3.5.1 Building Timed CSP Models

In our system, Timed CSP processes are in ASCII form, i.e., machine reachable Timed CSP which is used to enable machine readability. **HORAE** has a user-friendly editor to build Timed CSP models. The encoder **tcsp2clp** can automatically transform Timed CSP model in *.tcsp*-format into the CLP($\mathcal{R}$) *.clpr*-format by syntax rewriting.

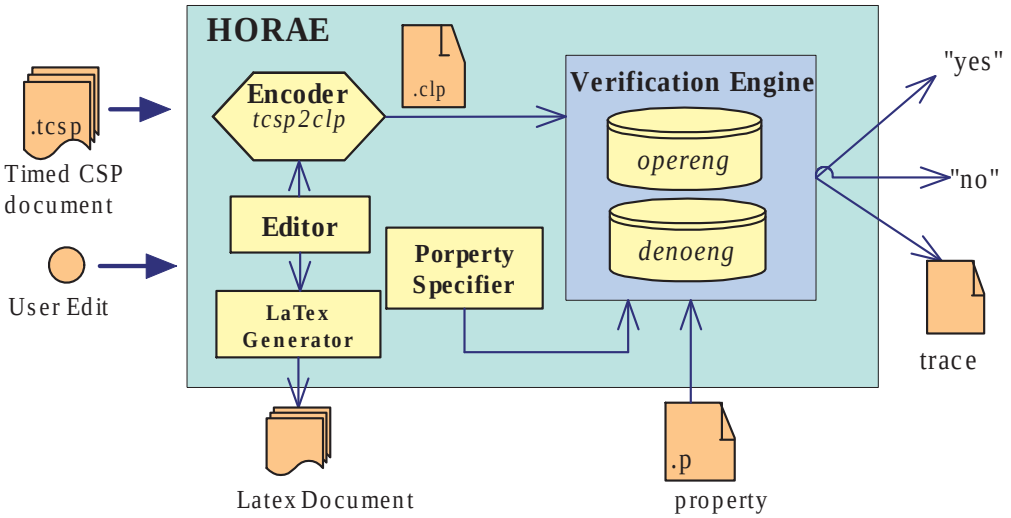

Figure 3.2: Overview of HORAE

3.5.2 Verifying Models

The verification engine is the core component of **HORAE**. The verification is performed by the engine which takes a Timed CSP model in *.clpr*-format from the encoder and a property as input. This engine has two modules, which are **opereng** and **denoeng**, building on the operational model and the denotational model respectively. Different kinds of properties are checked by different modules.

Timed CSP Semantics in CLP

In the verification engine, the semantics of Timed CSP processes are modeled in CLP. In our tool, both the operational semantics and denotational semantics are encoded.

Operational semantics is modelled by capturing both evolution relations and timed event transition relations of a process. In denotational semantics, we encoded the timed traces and timed failures model of Timed CSP, where a Timed CSP process is represented by a set of timed traces or a set of timed failures. Detailed theory and encoding of both semantics can be found in Section 3.2 and Section 3.3 respectively.

Property Specification and Verification

The module *Property Specifier* is used to specify three kinds of properties in a systematic way, which can later be verified by the verification engine.

Safety and Liveness Reachability properties are specified as: $Reachable(P, Q, \Psi) = N$ which tests whether N is a possible trace from process Q to P, with some constraint Ψ. Deadlock freeness of process P can be checked in $Deadlock(P)$.

Variable Bounds We can identify the lower or upper bound of a variable. For example, we can identify whether a given value of a time variable T is an upper or lower bound of duration of the execution of one process P to its subsequent process Q. These properties are specified as: $Upper(P,\ Q,\ time) = T$ or $Lower(P,\ Q,\ time) = T$. Similarly, the lower or upper bound of the duration between two events A and B in a process P can also be specified as $P :: Upper(A,\ B,\ time) = T$ or $P :: Lower(A,\ B,\ time) = T$.

3.6 Case Studies and Experiments

In this section, we compare our method to the mature model checker for CSP, namely (Failures-Divergence Refinement) [84] (version 2.78), in terms of flexibility as well as efficiency. We implement a prototype as a normal CLP($\mathcal{R}$) program. In the following, we demonstrate our experiments with three examples on a Unix system located at a Sunfire sever with 1GB user memory. Because FDR is designed for CSP, the quantitative timing aspects of the examples have been abstracted using discrete clock ticks first, then FDR is used to perform verification.

3.6.1 Timed Vending Machine

The specification of the timed vending machine is presented in Example 2.2.1. Figure 3.1 shows the timed vending machine model in CLP. This example is customized into a FDR

program (say P), in which the time-out operator is replaced with an external choice. The following are the properties verified:

- *tvm-1* Deadlock-freeness.
- *tvm-2* Trace timewise refinement:
  - in CLP, whether the process TVM is a trace timewise refinement of P.
  - in FDR, whether the process P is a trace refinement of TVM.
- *tvm-3* Whether there is such a case that coffee is selected while tea is dispatched.

3.6.2 Dining Philosopher

The classic dining philosopher example is also experimented. The specification is available in [51]. We implement this example with N philosophers and N forks. The following properties are experimented:

- *philosopherN-1* It is not deadlock-free.
- *philosopherN-2* No more than N+1/2 philosophers can eat at the same time.
- *philosopherN-3* It is possible that one philosopher eat all the time with the others starving. This property is checked with trace refinement.

3.6.3 The Railway Crossing

The railway crossing system is modelled and checked, which is complex enough to demonstrate a number of aspects of the modelling and verification of timed systems. The system consists of three components: a train, a gate and a controller. The gate should be up to allow traffic to pass when no train approaching and lowered to obstruct traffic when a train is coming. The controller monitors the approach of a train, and instructs the gate to be

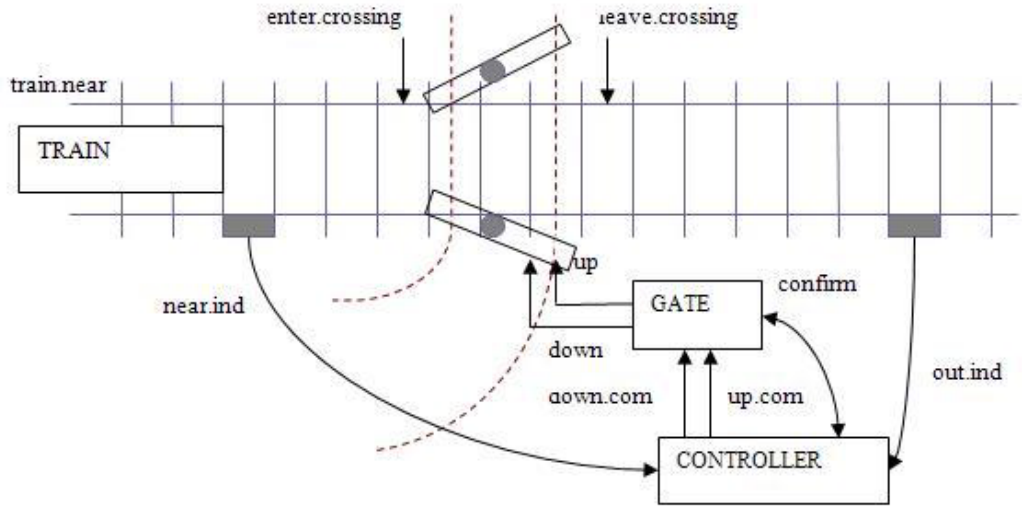

Figure 3.3: The Rail Way Control System

lowered within the appropriate time. The train is modelled abstractly with behaviors: nearing, entering and leaving the crossing. The Timed CSP modelling is as follows (originally presented in [89]):

$$
\begin{aligned}
&TRAIN \mathrel{\widehat{=}} \mu\, T \bullet trainnear \rightarrow nearind \overset{300}{\rightarrow} entercrossing \\
&\qquad \overset{20}{\rightarrow} leavecrossing \rightarrow outind \rightarrow T \\
&GATE \mathrel{\widehat{=}} \mu\, G \bullet downcom \overset{100}{\rightarrow} down \rightarrow confirm \rightarrow G \\
&\qquad \Box\ upcom \overset{100}{\rightarrow} up \rightarrow confirm \rightarrow G \\
&CONTROLLER \mathrel{\widehat{=}} \mu\, C \bullet outind \overset{1}{\rightarrow} upcom \rightarrow confirm \rightarrow C \\
&\qquad \Box\ nearind \overset{1}{\rightarrow} downcom \rightarrow confirm \rightarrow C \\
&CROSSING \mathrel{\widehat{=}} CONTROLLER\ {}_C||_G\ GATE \\
&SYSTEM \mathrel{\widehat{=}} TRAIN\ {}_T||_{C \cup G}\ CROSSING
\end{aligned}
$$

The timing information of the system is that: the train takes at least 5 minutes from triggering the *near.ind* sensor to reach the crossing; and at least 20 seconds to get across the crossing. The controller takes a negligible amount of time, say 1 second, from receiving a signal from a sensor to relaying the corresponding instruction to the gate. The gate

Property	**Goal in CLP**
deadlock-freeness	proc(system, P), tdeadlock(P, 0) $\models$ *false*
if train enters crossing, the gate must be down	proc(system, P), supersetp(P, X), last(X, entercrossing),filter(X, [up,down], X2), last(X2, up) $\models$ *false*
lower bound for a train passes the crossing is 320s	proc(system, P), dur(delay(nearind, _, _), eventprefix(outind, _), T1, T2),T2-T1<320 $\models$ *false*
if the gate is up, the train must have left the crossing	proc(system,P), superstep(P, X), not(not in([up, entercrossing, leavecrossing], X); after(X, leavecrossing, entercrossing))) $\models$ *false*
legal trace checking	proc(system, P),superstep(P, [trainnear, nearind, downcomm, down, confirm, entercrossing, leavecrossing, outind])$\models$ *true*

Table 3.2: Properties Verification

process takes 100 seconds to get itself into position following an instruction. A number of interesting properties can be formulated, evidenced in Table 5.1. The three properties selected for comparing our approach with FDR verification are:

- *railway-1* Deadlock-freeness
- *railway-2* Whether trace $\langle trainnear, nearind, downcomm, down, confirm, entercrossing, leavecrossing, outind\rangle$ is a legal trace or not.
- *railway-3* Whether the lower bound for a train passes the crossing is $320s$.

We summarize our results in Table 3.3. We ran the examples in both CLP($\mathcal{R}$) and FDR systems and we calculated the execution time of each property if the property is able to be checked in that system. From the table, we can see that most of our timing analysis performance are competitive with the well-known system, while in some cases, we are not

Assertion	**CLP($\mathcal{R}$)** *(sec)*	**FDR** *(sec)*	**Assertion**	**CLP($\mathcal{R}$)** *(sec)*	**FDR** *(sec)*
tvm-1	0.00	0.23	*phi3-1*	0.12	0.25
tvm-2	0.03	0.27	*phi3-2*	0.22	–
tvm-3	0.01	–	*phi3-3*	0.04	0.17
railway-1	0.25	0.25	*phi4-1*	0.84	0.28
railway-2	0.02	0.26	*phi4-2*	2.5	–
railway-3	0.32	–	*phi4-3*	0.1	0.3

Table 3.3: Experiment Results

so competitive. The important metric of our experiments is the flexibility. The results show that our reasoning method based on constraint solver.

3.7 Summary

In this chapter, we proposed a reasoning method for Timed CSP based on constraint logic. we have shown that event-based process algebra Timed CSP can be encoded in CLP by encoding both the operational and denotational semantics. This piece of work therefore broadened real-time systems which can be specified and verified by CLP. It is a solid foundation for the future research on extending Timed CSP and developing a reasoning engine for this extended Timed CSP specification.

This approach starts with a formal translation of Timed CSP syntax to CLP relations, where a library of all Timed CSP operators is built. We have also developed a collection of supplementary rules for encoding operational semantics of each operator into CLP rules. These serve as a solid foundation in our verification system. We can rigorously carry out verification of Timed CSP models with a high level of automation. We investigated a wide range of properties that may be proved based on constraint solving, namely CLP($\mathcal{R}$), for instance we showed that using a unique interpretation, traditional safety and liveness can

be proved effectively as well as properties such as lower or upper bound of a variable and refinement.

Chapter 4

Timed Planning

As demonstrated in last chapter, Timed CSP is a powerful language to model real-time reactive systems. However, limited by the expressive power, it lacks of support for stating system requirements which constraints all behavioral traces of given processes, for example, *deadline* and execution time of a process, time-related constraints among events which are common requirements for many real time systems and etc. To increase the expressiveness of Timed CSP, we propose an extension of Timed CSP, which named Timed Planning. Timed Planning specification extends Timed CSP with the capability of stating complicated timing behaviors for processes and events to model and verify complex compositional real-time systems. A Timed Planning model is made up of a compositional timed process and a set of constraints over processes, events and the data variables which are the requirements that the process should satisfy. In this proposed formalism, each process is associated with a set of localized timing/untiming requirements with keyword WHERE which can be specified in a compositional way. The full syntax and operational semantics of Timed Planning language are formally defined in this chapter.

A reasoning mechanism for the Timed Planning is hence developed based on Constraint Logic Programming (CLP) by extending the reasoning engine for Timed CSP. Both syntax and semantics of Timed Planning are formally translated into CLP. The operational semantics

is encoded to CLP, where a set of global and local variables need to be captured during the execution. Feasibility checking and various property verification can be applied to check systems modeled in Timed Planning.

4.1 Syntax of Extended Timed CSP

In the extended Timed CSP specification, a process is extended with an optional WHERE clause, which consists of a (first order) predicate over a predefined set of time variables.

4.1.1 Timed Variables

To capture the important timing information of a process, we introduce three time point variables: START, END, and TES for processes and one time point variable ENGAGE for events, which are as explained as follows.

Starting and Ending Time of a Process

One important timing information of a process is when it is started and when it is ended. Given a process P, the variable P.START (P.END) denotes the exact starting (ending) time of process P. More specifically, P.START captures the starting time of a process P when its first event is enabled or when the "WAIT d" process is enabled if P starts with a WAIT process.

P.END is the ending time of the process P. If the process is terminating, P.END is the engage time of the termination event $\checkmark$. If the process is non-terminating, then P.END $= \infty$. Naturally, condition P.END $\geq P$.START holds all the time. Using the two variables, a *deadline* property (a task must be accomplished within a certain time) is expressed as P WHERE P.END $- P$.START $\leqslant d$ where d is a constant ($d \in \mathbb{R}^+$), if and only if this process is terminating.

Example 4.1.1 Kate needs to attend a meeting which starts at 10:00 and lasts less than two hours. She will go home after the meeting.

$$\begin{aligned} \mathit{Kate} \mathrel{\widehat{=}} & \mathit{arrive} \rightarrow \mathit{Meeting};\ \mathit{gohome} \rightarrow \textsc{Skip} \\ & \textsc{Where}\ \mathit{arrive}.\textsc{Engage} \leqslant 10 \wedge \mathit{Meeting}.\textsc{Start}{=}10 \\ & \wedge\ \mathit{Meeting}.\textsc{End} - \mathit{Meeting}.\textsc{Start} \leqslant 2 \end{aligned}$$

where we write 10 to represent 10:00 for simplicity. Meaning of ENGAGE is explained below. **end**

Engagement Time of Events

In many scenarios, there are requirements on an event to perform at some expected time, for example attending a meeting at 10am or printing the job within 15 minutes after receiving this job. A time variable ENGAGE is attached to an event e to denote the exact time when e is engaged, in the form of $e.\textsc{Engage}$.

In an evaluation, event e is likely to be engaged more than once, for example, $P \mathrel{\widehat{=}} a \xrightarrow{2} P$. The variable $e.\textsc{Engage}_i$ is introduced to denote the i_{th} occurrence of e in an evaluation. For instance, $P \mathrel{\widehat{=}} a \xrightarrow{2} P$, one possible evaluation is $\langle (0, a), (3, a), (5, a) ... \rangle$. In this evaluation, the engage time of event a is stored in $a.\textsc{Engage}_i$, where $i = 1...N$: $(a.\textsc{Engage}_1 = 0, a.\textsc{Engage}_2 = 3, a.\textsc{Engage}_3 = 5, \ldots)$.

For event e engaged more than once in a trace, $e.\textsc{Engage}$ is the set of all occurrences of event e in the process it is attached to. More specifically, $e.\textsc{Engage}$ is the set of $e.\textsc{Engage}_i$, $e.\textsc{Engage}_i$ is the element of set $e.\textsc{Engage}$, i.e.,

$$\begin{aligned} & e.\textsc{Engage} = \{e.\textsc{Engage}_i\} \\ & e.\textsc{Engage}_i \in e.\textsc{Engage} \end{aligned}$$

For simplicity, we define a set of rules to eliminate the $\forall\, i$ to be used in our constraints. For instance, to specify the constraint that event e must be engaged before time t, expression $e.\textsc{Engage} \leqslant t$ is used instead of $\forall\, i \bullet e.\textsc{Engage}_i \leqslant t$, if this process is terminating.

$$
\begin{array}{ll}
e.\textsc{Engage} \leqslant t & \equiv \forall\, i \bullet e.\textsc{Engage}_i \leqslant t \\
0 \leqslant e1.\textsc{Engage}_i - e2.\textsc{Engage}_i \leqslant t & \equiv \forall\, i \bullet 0 \leqslant e1.\textsc{Engage}_i - e2.\textsc{Engage}_i \leqslant t \\
0 \leqslant e1.\textsc{Engage} - e2.\textsc{Engage} \leqslant t & \equiv \forall\, i, j \bullet 0 \leqslant e1.\textsc{Engage}_i - e2.\textsc{Engage}_j \leqslant t,
\end{array}
$$

Constraint $e1.\textsc{Engage}_i - e2.\textsc{Engage}_i \leqslant t$ is exactly the same as $\forall\, i \bullet e1.\textsc{Engage}_i - e2.\textsc{Engage}_i \leqslant t$, which means that each time after $e2$ is engaged, $e1$ must be engaged within t time units. While constraint $e1.\textsc{Engage} - e2.\textsc{Engage} \leqslant t$ means $\forall\, i, j \bullet e1.\textsc{Engage}_i - e2.\textsc{Engage}_j \leqslant t$, which specifies the requirement that $e1$ is not allowed to be engaged after t time units with any occurrence of $e2$.

Timed Event Set

In addition, the variable $P.\textsc{Tes}$ is introduced to capture the evaluation of a process P up to the current time, where Tes stands for *T*imed *E*vent *S*et. Tes is a set of *timed event*s. A *timed event* is a pair ($e \times \mathbb{R}^+$, where $e \in \Sigma$[1]) consisting of a time and an event engage time variable.

Tes is used to record the engage time of all events engaged so far, which can be viewed as a history of the execution. Traces of the specific evaluation are able to be retrieved from Tes. We can also retrieve other information we are interested in from the set Tes. The following example illustrates a constraint concerning the number of occurrences of events in P.

Example 4.1.2 In a restaurant, the staff in the kitchen cook and supply food to the counter, the staff at the counter takes orders and delivers food to the customers.

$$
\begin{array}{ll}
\mathit{Kitchen} & \widehat{=}\ \mathit{cook} \rightarrow \mathit{supply} \rightarrow \mathit{Kitchen} \\
\mathit{Counter} & \widehat{=}\ \mathit{order} \stackrel{30}{\rightarrow} \mathit{serve} \rightarrow \mathit{Counter} \\
 & \quad \textsc{Where}\ \mathit{serve}.\textsc{Engage} - \mathit{order}.\textsc{Engage} \leqslant 60 \\
\mathit{Mcd} & \widehat{=}\ \mathit{Kitchen} \parallel \mathit{Counter} \\
 & \quad \textsc{Where}\ \textsc{Tes} \downarrow \mathit{supply} \geq \textsc{Tes} \downarrow \mathit{serve} \wedge \textsc{Tes} \downarrow \mathit{supply} - \textsc{Tes} \downarrow \mathit{serve} \leqslant 10
\end{array}
$$

[1] Σ is the universal set of events.

where TES $\downarrow$ *supply* is the number of occurrences of event *supply* in the timed event set TES up to the current time. The WHERE clause guarantees for each order, there is available food to be delivered. **end**

4.1.2 Syntax of Timed Planning

In this section, we present the formal syntax for Timed Planning process. A Timed Planning process is a Timed CSP process attached with a optional WHERE predicate which is defined by the following syntax tree.

$$
\begin{aligned}
P(x_1, ..., x_n) ::= & \ \mathit{ProcExp}\ [\textsc{Where}\ \mathit{WherePred}] \\
\mathit{WherePred} ::= & \ \mathit{WherePred} \wedge \mathit{WherePred} \\
| & \ \mathit{WherePred} \vee \mathit{WherePred} \\
| & \ \mathit{WherePred} \Leftrightarrow \mathit{WherePred} \\
| & \ \mathit{WherePred} \Rightarrow \mathit{WherePred} \\
| & \ \neg\, \mathit{WherePred} \mid \mathit{true} \mid \mathit{false} \\
| & \ \mathit{WhereExpr} \sim \mathit{WhereExpr}, \\
& \qquad\qquad \sim \ \in \{<, \leqslant, =, >, \geq\} \\
\\
\mathit{WhereExpr} ::= & \ [\mathit{Name}.]\textsc{Start} \mid [\mathit{Name}.]\textsc{End} \\
| & \ [\mathit{Name}.]\textsc{Tes} \\
| & \ \mathit{Name}.\textsc{Engage} \mid \mathit{Name}.\textsc{Engage}_i \\
| & \ \mathit{WhereExpr} \odot \mathit{WhereExpr}, \odot \in \{+, -\} \\
| & \ \mathit{WhereExpr} \odot \mathbb{R}, \odot \in \{*, /\} \\
| & \ (\mathit{WhereExp}) \mid \mathbb{R}
\end{aligned}
$$

where P is the process name, $x_1, ..., x_n$ is an optional list of process parameters and *ProcExp* is a Timed CSP process expression. Each process can be attached with an optional *WherePred* with keyword WHERE. Process P without *WherePred* is exactly the same as a Timed CSP process.

Name is a sequence of characters starting an alphabet, i.e., an event or a process. Note that if *Name* is missing, it defaults to the process name on the left hand side. To differentiate events of the same name in different processes, we write $P.a$ to denote the event a in P whenever necessary. By using the syntax defined above, common Timed Planning requirements can be specified easily.

Example 4.1.3 Consider a printer system, which has a receiver and two printers. The receiver receives jobs and the two printers print jobs.

$$
\begin{aligned}
&\mathit{Receiver} \mathrel{\widehat{=}} \mathit{accept} \rightarrow \mathit{Receiver}\\
&\mathit{Printer}1 \mathrel{\widehat{=}} \mathit{print} \overset{30}{\rightarrow} \mathit{print\_finish} \rightarrow \mathit{Printer}1\\
&\qquad \textsc{Where}\mathit{print\_finish}.\textsc{Engage}_i - \mathit{print}.\textsc{Engage}_i \leqslant 120\\
&\mathit{Printer}2 \mathrel{\widehat{=}} \mathit{print} \overset{30}{\rightarrow} \mathit{print\_finish} \rightarrow \mathit{Printer}2\\
&\qquad \textsc{Where}\mathit{print\_finish}.\textsc{Engage}_i - \mathit{print}.\textsc{Engage}_i \leqslant 120\\
&\mathit{PrintSys} \mathrel{\widehat{=}} \mathit{Receiver} \;|||\; \mathit{Printer}1 \;|||\; \mathit{Printer}2\\
&\qquad \textsc{Where}\textsc{Tes} \downarrow \mathit{print} \leqslant \textsc{Tes} \downarrow \mathit{accept}
\end{aligned}
$$

Process *Receiver* receives printing jobs and processes *Printer*1 and *Printer*2 are the two printers printing documents. For each document printing process, it should be finished within 2 minutes (120 seconds) and it must take at least 30 seconds for the whole process. Since receiver and the two printers are working interleavingly, we need to make sure that for each printing, there must be at least one unprinted jobs. Constraint $\textsc{Tes} \downarrow \mathit{print} \leqslant \textsc{Tes} \downarrow \mathit{accept}$ is used to capture this idea, where $\textsc{Tes} \downarrow \mathit{print}$ is the number of occurrences of event *print* in the event set Tes. □**end**

$\textsc{Tes} \downarrow e$ is to count the number of event e in an execution that has been engaged so far. It is a unique feature of Timed Planning where Timed CSP or Timed Automata are unable to achieve. To capture different aspects of the execution, we define a set of build-in libraries which can be directly used in the where predicates.

- $\textsc{Tes} \downarrow e$: the number of occurrences of event e in the timed event set Tes.
- *first*(Tes): The first event engaged in Tes.
- *last*(Tes): The last event engaged in Tesup to the current execution.

Since Tes is a set of pairs $(e \times \mathbb{R}^+)$, $\textsc{Tes} \downarrow e$ is easily calculated by counting the number of appearances of e in Tes which have been engaged so far. *first*(Tes) is retrieved by finding the pair (e, T) with the smallest T in Tes, while *last*(Tes) is to find the largest T. Users are free to add their own functions to the libraries.

Common requirements for a system can be easily specified. For instance, the *deadline* of a process, order of events, separation time between events, etc as follows:

1. Process P must be finished within time d:

   $P.\text{END} - P.\text{START} \leqslant d$

2. Process P must be finished before time d:

   $P.\text{END} \leqslant d$

3. Max separation time between two events e_1, e_2 is d:

   $e_2.\text{ENGAGE} - e_1.\text{ENGAGE} \leqslant d$

4. e_2 must be engaged before e_1:

   $e_1.\text{ENGAGE} - e_2.\text{ENGAGE} \leqslant 0$

Data types

The Timed Planning specification supports a wide range of primitive data types, including integers, real numbers, boolean values and lists. We can define global variables or process parameters of these types. To be specific, the data types we are supporting are: $\mathbb{R}$ (Real Number), $\mathbb{N}$ (Integers), $\mathbb{B}$ (Boolean) and $T[]$ (a list of elements of type T where $T \in \{\mathbb{R}, \mathbb{N}, \mathbb{B}\}$).

4.1.3 Healthiness Conditions

As illustrated in the last section, each process has a WHERE clause, which restricts the set of timed traces of the process. The constraints associated with a process are divided into two groups, namely the explicit ones defined in the WHERE clause and implicit ones. The implicit constraints should always be satisfied, which is a set of healthiness conditions of processes. The following are two examples of such healthiness conditions. The complete list of healthiness conditions can be found in Appendix A.

- For every event e in process P, e must be engaged between the starting time and ending time of P. Let $\sum P$ be the alphabet of P.

$$\forall\, e : \sum P \bullet P.\textsc{Start} \leqslant e.\textsc{Engage} \wedge e.\textsc{Engage} \leqslant P.\textsc{End}$$

- Let P_i be a sub-process of P, written as $P_i \preccurlyeq P$. The starting time of P_i must be greater than or equal to the starting time of P and its ending time must be less than or equal to that of P.

$$\forall\, P, P_i \bullet P_i \preccurlyeq P \Rightarrow P.\textsc{Start} \leqslant P_i.\textsc{Start} \wedge P_i.\textsc{End} \leqslant P.\textsc{End}$$

4.2 Operational Semantics

In the previous section, the syntax Timed Planning processes is illustrated. In this section, we formally define the semantics of the new specification. An operational semantics provides a way of interpreting a language by stepping through executions of programs written in that language. It describes an operational understanding of the language. The operational semantics of Timed CSP is precisely defined in Schneider [90] by using the combination of two relations: event transition and evolution. The semantic model consists of three components: the event and timed transitions which are inherited from Timed CSP, a WHERE predicate which must be satisfied by this model, and an *Timed stamped set*.

A *Timed stamped set* (Tss) is a record of an execution, which is to store all the values of the timed variables introduced in last section, namely START, END, ENGAGE, $\textsc{Engage}_i$ and TES. It consists of a set of process related time stamps, namely the starting and ending times of processes, and a *timed event set* (TES) which is a set of timed events. TES is a subset of *Tss*: $\textsc{Tes} \subseteq \mathit{Tss}$. A *timed event* is a pair drawn from $e \times \mathbb{R}^+$ where $e \in \Sigma$, consisting of a time and an event engage time value.

We define the state of a process as a quadruple $\langle P, t, W, \mathit{Tss}\rangle$ where P is the process, t is the current time, W is the WHERE predicate and *Tss* is the *timed stamped set* of the model. *Tss* keeps value for all variables. At each transition, an evaluation of the system

requirement W is performed. If the current state satisfies the requirement, the transition can be enabled, otherwise not. The operational semantics of Timed Planning is defined as follows.

Definition 4 *The operational semantics of the extended Timed CSP specification is a timed transition where the state is a quadruple* $\langle P, t, W, Tss\rangle$*, and event transitions and evolution transitions are defined by the rules:*

- $\langle P, t, W, Tss\rangle \xrightarrow{a} \langle P', t, W, Tss'\rangle$ *where*
  $a \neq \checkmark \wedge \exists\, i : \mathbb{N} \bullet Tss \cup \{(a.\textsc{Engage}_i, t), (P.\textsc{Start}, t)\} \vDash W$
- $\langle P, t, W, Tss\rangle \xrightarrow{\checkmark} \langle P', t, W, Tss'\rangle$ *where*
  $Tss \cup \{(P.\textsc{End}, t)\} \vDash W$
- $\langle P, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \langle P', t + d, W, Tss'\rangle$ *where*
  $d > 0 \wedge Tss \cup \{(P.\textsc{Start}, t)\} \vDash W$

where P' *is the subsequent process of* P *by involving either a event transition* $(\rightarrow)$ *or a timed transition* $(\rightsquigarrow)$*.* Tss' *is a timed stamped set with updated* Engage, $\textsc{Engage}_i$, Start, End *and* Tes *variables. The* $\xrightarrow{a}$ *represents an event transition, whereas* $\overset{d}{\rightsquigarrow}$ *is a timed transition.* $\exists\, i : \mathbb{N} \bullet Tss \cup \{(a.\textsc{Engage}_i, t),\ (P.\textsc{Start}, t)\} \vDash W$ *is an evaluation of the current state, which is used to check whether the current Tss fulfills the system requirements* W *or not.* □

In the operational semantics, we define both event transition and timed transition relations for all primary and compositional operators in Timed Planning specification.

Stop

In Timed Planning, process Stop is the same as it is in Timed CSP. It remains unable to perform any internal or external events which must still be in the same state after any delay has occurred. The inference rule for Stop is defined as follows:

$$\frac{}{\langle \textsc{Stop}, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \langle \textsc{Stop}, t+d, W, Tss\rangle}$$

This states that process STOP can allow any amount of time d to elapse, and that the resulting state will still be STOP. Tss remains unchanged.

Skip

The Timed CSP process SKIP is the immediately terminating process. The inference rules for the operational semantics of SKIP are defined as follows:

$$\frac{Tss \cup \{(\checkmark.\textsc{Engage}, t)\} \models W}{\langle \textsc{Skip}, t, W, Tss\rangle \xrightarrow{\checkmark} \langle \textsc{Stop}, t, W, Tss \cup \{(End, t)\}\rangle}$$

$$\frac{}{\langle \textsc{Skip}, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \langle \textsc{Skip}, t+d, W, Tss\rangle}$$

Event Prefix: a → Q

The CSP expression a → P describes a process which is prepared to engage in the event a, after which it will behave as P. In Timed Planning, it remains in its initial state until event a is able to be performed at the current environment and requirements. Alternatively, some time may elapse without the event a being accepted. In this case, the process remains in the same state, patiently maintaining the offer. These two possibilities are described by the two following transitions.

$$\frac{Tss \cup \{(a.\textsc{Engage}_i, t)\} \models W}{\langle a \rightarrow Q, t, W, Tss\rangle \xrightarrow{a} \langle Q, t, W, Tss \cup \{(a.\textsc{Engage}_i, t)\}\rangle} [\ Tes \downarrow a = i-1\ ]$$

$$\frac{}{\langle a \rightarrow Q, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \langle Q, t+d, W, Tss\rangle}$$

Timeout: $Q_1 \triangleright \{d\}\, Q_2$

The timeout operator introduces a way of describing time-sensitive process behavior. Initially process Q_1 is available, which means that the control is with process Q_1 when execution begins. If Q_1 performs some external event before d units of time have elapsed, providing that this event engaging at the current time satisfies the system requirements, then the timeout choice is resolved in favor of Q_1. In this case, Q_2 is discarded without ever being made available. Q_1 performing an internal event would not resolve the choice. If Q_1 does not perform any visible event in the first d time units, the process would pass control to process Q_2, where the timeout occurs.

$$\frac{Tss \cup \{(a.\textsc{Engage}_i, t)\} \models W \qquad Q_1 \xrightarrow{a} Q_1'}{\langle Q_1 \overset{d}{\triangleright} Q_2, t, W, Tss\rangle \xrightarrow{a} \langle Q_1', t, W, Tss \cup \{(a.\textsc{Engage}_i, t)\}\rangle} [\ Tes \downarrow a = i - 1\ ]$$

$$\frac{Q_1 \xrightarrow{\tau} Q_1'}{\langle Q_1 \overset{d}{\triangleright} Q_2, t, W, Tss\rangle \xrightarrow{\tau} \langle Q_1' \overset{d}{\triangleright} Q_2, t, W, Tss\rangle}$$

$$\frac{}{\langle Q_1 \overset{0}{\triangleright} Q_2, t, W, Tss\rangle \xrightarrow{\tau} \langle Q_2, t, W, Tss\rangle}$$

$$\frac{Q_1 \overset{d'}{\rightsquigarrow} Q_1'}{\langle Q_1 \overset{0}{\triangleright} Q_2, t, W, Tss\rangle \overset{d'}{\rightsquigarrow} \langle Q_1' \overset{d-d'}{\triangleright} Q_2, t + d', W, Tss\rangle}$$

External Choice: $Q_1 \,\square\, Q_2$

The external choice operator offers a choice between processes, which is resolved at the instant the first visible event occurs, in favor of the process which performs it. In Timed Planning, the choice is resolved only by an external event of one of participating process. The choice is preserved when any participating process performing an internal event. Since time passes for both processes are at the same rate, they both participate in any evolution. The operational semantics of external choice operator are illustrated by the following rules.

$$\frac{\begin{array}{l} Tss \cup \{a.\textsc{Engage}_i, t\} \models W \\ Q_1 \xrightarrow{a} Q_1' \end{array}}{\begin{array}{l} \langle Q_1 \,\Box\, Q_2, t, W, Tss\rangle \xrightarrow{a} \\ \langle Q_1' \,\Box\, Q_2, t, W, Tss \cup \{(a.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ Tes \downarrow a = i - 1\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{a.\textsc{Engage}_i, t\} \models W \\ Q_2 \xrightarrow{a} Q_2' \end{array}}{\begin{array}{l} \langle Q_1 \,\Box\, Q_2, t, W, Tss\rangle \xrightarrow{a} \\ \langle Q_1 \,\Box\, Q_2', t, W, Tss \cup \{(a.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ Tes \downarrow a = i - 1\ ]$$

$$\frac{Q_1 \xrightarrow{\tau} Q_1'}{\langle Q_1 \,\Box\, Q_2, t, W, Tss\rangle \xrightarrow{\tau} \langle Q_1' \,\Box\, Q_2, t, W, Tss\rangle}$$

$$\frac{Q_2 \xrightarrow{\tau} Q_2'}{\langle Q_1 \,\Box\, Q_2, t, W, Tss\rangle \xrightarrow{\tau} \langle Q_1 \,\Box\, Q_2', t, W, Tss\rangle}$$

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1' \qquad Q_2 \overset{d}{\rightsquigarrow} Q_2'}{\langle Q_1 \,\Box\, Q_2, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \langle Q_1' \,\Box\, Q_2', t + d, W, Tss\rangle}$$

Parallel: $Q_1 \ {}_X||_Y \ Q_2$

The operational semantics of the alphabetized parallel composition operator $Q_1 \ {}_X||_Y \ Q_2$ are illustrated in the following rules.

$$\frac{\begin{array}{l} Tss \cup \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\} \models W \\ Q_1 \xrightarrow{e} Q_1' \end{array}}{\begin{array}{l} [e \in X \cup \{\tau\} \setminus Y, Tes \downarrow e = i - 1] \\ \langle Q_1 \ {}_X||_Y \ Q_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle Q_1' \ {}_X||_Y \ Q_2, t, W, Tss \cup \\ \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ r1\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\} \models W \\ Q_2 \xrightarrow{e} Q_2' \end{array}}{\begin{array}{l} [e \in Y \cup \{\tau\} \setminus X, Tes \downarrow e = i - 1] \\ \langle Q_1 \ {}_X||_Y \ Q_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle Q_1 \ {}_X||_Y \ Q_2', t, W, Tss \cup \\ \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ r2\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\} \models W \\ Q_1 \xrightarrow{e} Q_1' \quad Q_2 \xrightarrow{e} Q_2' \end{array}}{\begin{array}{l} [e \in X \cap Y - \{\checkmark\}, Tes \downarrow e = i - 1] \\ \langle Q_1 \ {}_X||_Y \ Q_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle Q_1' \ {}_X||_Y \ Q_2', t, W, Tss \cup \\ \{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ r3\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{(\textsc{Start}, t), (\checkmark.\textsc{Engage}, t)\} \models W \\ Q_1 \xrightarrow{\checkmark} Q_1' \qquad Q_2 \xrightarrow{\checkmark} Q_2' \end{array}}{\begin{array}{l} \langle Q_1 \ {}_X||_Y \ Q_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle Q_1' \ {}_X||_Y \ Q_2', t, W, Tss \cup \\ \{(\textsc{Start}, t), (\textsc{End}, t)\}\rangle \end{array}} [\ r4\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{(\textsc{Start}, t)\} \models W \\ Q_1 \overset{d}{\rightsquigarrow} Q_1' \qquad Q_2 \overset{d}{\rightsquigarrow} Q_2' \end{array}}{\begin{array}{l} \langle Q_1 \ {}_X||_Y \ Q_2, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \\ \langle Q_1' \ {}_X||_Y \ Q_2', t + d, W, Tss \cup \{(\textsc{Start}, t)\}\rangle \end{array}} [\ r5\ ]$$

The first two rules state that either of the components (Q_1 or Q_2) may engage an event as long as the event is not shared if and only if the evaluation on whether the current Tss appended with $\{(\textsc{Start}, t), (e.\textsc{Engage}_i, t)\}$ still satisfies process requirement W is true. Tss will be updated to $Tss' = Tss \cup \{(\textsc{Start}, t),\ (e.\textsc{Engage}_i, t)\}$. Rule $r3$ states that a shared event can be engaged simultaneously by both components as long as the event satisfies the requirements. Rule $r4$ is a special case for the third rule, whereas the event is the ✓ which is a special event used purely to denote termination. A new pair (End, t) is added to the Tss and hence checked. Rule $r5$ says that the composition may allow time elapsing when both the components do.

Interleaving: $Q_1 \ ||| \ Q_2$

The introduction of time to the interleaving operator is entirely similar to the approach taken for the parallel composition. In an interleaved combination each internal or external

event is performed by precisely only one of the components if and only if the evaluation on the current Tss with this event satisfied the system requirements, while the other component makes no progress at all. The passage of time occurs at the same rate in both processes, so they must agree on timed transitions. The transitions reflecting operational semantics of interleaving operator are defined as follows:

$$\frac{\begin{array}{l} Tss \cup \{(e.\textsc{Engage}_i, t)\} \models W \\ Q_1 \xrightarrow{e} Q_1' \end{array}}{\begin{array}{l} \langle Q_1 \mid\mid\mid Q_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle Q_1' \mid\mid\mid Q_2, t, W, Tss \cup \{(e.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ Tes \downarrow e = i - 1\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{(e.\textsc{Engage}_i, t)\} \models W \\ Q_2 \xrightarrow{e} Q_2' \end{array}}{\begin{array}{l} \langle Q_1 \mid\mid\mid Q_2, t, W, Tss\rangle \xrightarrow{e} \\ \langle Q_1 \mid\mid\mid Q_2', t, W, Tss \cup \{(e.\textsc{Engage}_i, t)\}\rangle \end{array}} [\ Tes \downarrow e = i - 1\ ]$$

$$\frac{\begin{array}{l} Tss \cup \{(\checkmark.\textsc{Engage}, t)\} \models W \\ Q_1 \xrightarrow{\checkmark} Q_1' \qquad Q_2 \xrightarrow{\checkmark} Q_2' \end{array}}{\langle Q_1 \mid\mid\mid Q_2, t, W, Tss\rangle \xrightarrow{\checkmark} \langle Q_1' \mid\mid\mid Q_2', t, W, Tss \cup \{(\textsc{End}, t)\}\rangle}$$

$$\frac{Q_1 \overset{d}{\rightsquigarrow} Q_1' \qquad Q_2 \overset{d}{\rightsquigarrow} Q_2'}{\langle Q_1 \mid\mid\mid Q_2, t, W, Tss\rangle \overset{d}{\rightsquigarrow} \langle Q_1' \mid\mid\mid Q_2', t + d, W, Tss\rangle}$$

Example 4.2.1 Take a printer process as an example. After the printer accepts a job, it needs to print this job within 30 to 60 seconds. Assume the process starts at time 0.

$$\begin{array}{l} Printer \mathrel{\widehat{=}} accept \xrightarrow{30} print \rightarrow Printer \\ \qquad \textsc{Where}\ print.\textsc{Engage}_i\text{-}accept.\textsc{Engage}_i \leqslant 60 \end{array}$$

W is the constraint $print.\textsc{Engage}_i\text{-}accept.\textsc{Engage}_i \leqslant 60$. The following is one possible execution sequence.

$\langle accept \overset{30}{\rightarrow} print \rightarrow Printer, 0,$
$\quad \{print.\text{ENGAGE-}accept.\text{ENGAGE} \leqslant 60\}, \{(\text{START}, 0)\}\rangle \overset{10}{\rightsquigarrow}$
$\langle accept \overset{30}{\rightarrow} print \rightarrow Printer, 10,$
$\quad \{print.\text{ENGAGE-}accept.\text{ENGAGE} \leqslant 60\}, \{(\text{START}, 0)\}\rangle \overset{accept}{\rightarrow}$
$\langle \text{STOP} \overset{30}{\triangleright} print \rightarrow Printer, 10, \{print.\text{ENGAGE-}accept.\text{ENGAGE} \leqslant 60\},$
$\quad \{(\text{START}, 0), (accept.\text{ENGAGE}_1, 10)\}\rangle \overset{30}{\rightsquigarrow}$
$\langle print \rightarrow Printer, 40, \{print.\text{ENGAGE} - accept.\text{ENGAGE} \leqslant 60\},$
$\quad \{(\text{START}, 0), (accept.\text{ENGAGE}_1, 10)\}\rangle \overset{20}{\rightsquigarrow}$
$\langle print \rightarrow Printer, 60, \{print.\text{ENGAGE} - accept.\text{ENGAGE} \leqslant 60\},$
$\quad \{(\text{START}, 0), (accept.\text{ENGAGE}_1, 10)\}\rangle \overset{print}{\rightarrow}$
$\langle Printer, 60, \{print.\text{ENGAGE} - accept.\text{ENGAGE} \leqslant 60\}$
$\quad \{(\text{START}, 0), (accept.\text{ENGAGE}_1, 10), (print.\text{ENGAGE}_1, 60)\}\rangle$
$\ldots$

In this execution, event *accept* is firstly engaged at 10, we insert $(accept.\text{ENGAGE}_i 10)$ to *Tss*. It is not likely for event *print* to be firstly engaged after 40 seconds while it is enabled, where $print.\text{ENGAGE}_i$ will be greater than 70, since it is guarded by $print.\text{ENGAGE}_i$-$accept.\text{ENGAGE}_i \leqslant 60$. **end**

4.3 Modeling Timed Planning in CLP

4.3.1 Encoding Extended Timed CSP in CLP

The very initial step is to encode the extended Timed CSP models in to CLP rules. This step is automatically done by syntax rewriting. A process *"Proc* WHERE *WherePred"* is encoded to a relation *tproc(N, P, W)* in CLP. *tproc(N, P, W)* consists of three parts, where N is the name of this process, P is the CLP representation of *Proc* and W represents the *WherePred*. The syntax translation consists of two parts, process operators translation and *Where* clause translation.

All operators of the extended specification which inherited from Timed CSP are encoded into CLP rules in a compositional way. A library of all operator translation is built. For example:

- $a \rightarrow$ SKIP : `eventprefix(a, skip)`
- $a \stackrel{t}{\rightarrow}$ SKIP : `delay(a, skip, t)`
- $P_1 \,|||\, P_2$: `interleaving(P1,P2)`
- $P_1;\ P_2$: `sequential(P1,P2)`

In our library, $eventprefix(A, P)$ is defined to denote a process $A \rightarrow P$. $delay(A, P, T)$ is the CLP form of operator $A \stackrel{T}{\rightarrow} P$ in Timed CSP. $interleaving(P1, P2)$ is to represent operator $P_1 \,|||\, P_2$ where $P1$ and $P2$ are the CLP formate of process P_1 and P_2. Relations $sequential/2$ is to represent a sequential operator ";".

For each process, a WHERE clause *WherePred* is encoded into W in CLP. W is a list of the form $[W1, W2, ..., Wn]$. Before we encode *WherePred* into a CLP list, we need to convert *WherePred* into conjunctive normal form (CNF), which is a conjunction of Horn clauses. Logically, $W = W1 \wedge W2 \wedge ... \wedge Wn$ where each W_i is a Horn clause in the form of $or(Wi_1, Wi_2)$, $not(Wi_k)$ or an atom. If there are no WHERE predicates defined for this process, $W = []$. The syntax encoding of task *Kitchen* (Example 4.1.2) is as follows.

$tproc(kitchen, eventprefix(cook, eventprefix(supply, kitchen)), [\,]).$
$tproc(counter, delay(order, eventprefix(serve, counter), 30),$
$\qquad [leq(engage(serve), engage(order))]).$
$tpoc(mc, parallel(kitchen, counter),$
$\qquad [geq(number(tr(mc), supply), number(tr(mc), order)),$
$\qquad leq(minus(number(tr(mc), supply), number(tr(mc), order)), 5)]).$

where relations $leq, geq, minus, number$ are built-in predicates defined in our library, which represent $\leqslant, \geq, -, \downarrow$ respectively.

4.3.2 Encoding Semantics in CLP

Having defined the corresponding CLP syntax for the extended Timed CSP specifications, we devote the rest of this section to describe how to embed the operational semantics into

CLP rules. A relation of the form *tpos*(P_1, T_1, E_1, M, P_2, T_2, E_2) is used to denote the *t*imed *p*rotocol *o*perational *s*emantics, by capturing both event transition relations and evolution relations with a set of constraints. Informally speaking, $tpos(P_1, T_1, E_1, M, P_2, T_2, E_2)$ returns true if the process P_1 evolves to P_2 through either a time evolution, i.e., let $T_2 - T_1$ time units elapse (so that $M = []$), or an event transition by engaging an event e instantly ($M = e$), as long as both transitions satisfy the WHERE requirements stored in E_1. After this transition relation, the local environment might change to E_2 by adding more predicates. E_1 (and E_2) is the environment of the system, which consists not only the WHERE predicates, but also the current values of the variables appeared in the WHERE predicates.

We define the $tpos/7$[2] relation for each and every operator of Timed CSP according to the semantics presented previously in Section 4.2.

$tpos(stop, T1, E, [], stop, T2, E) :- D >= 0, T2 = T1 + D.$
$tpos(skip, T, E1, [termination], stop, T, E2)$
$\quad :- sat(E, termination, T, E2).$
$tpos(skip, T1, E1, [], skip, T2, E2)$
$\quad :- D >= 0, T2 = T1 + D, sat(E1, T1, E2).$

The only transition for process STOP is time elapsing. Process SKIP may choose to wait some time before engaging the ✓ event. We use *termination* to denote this special event in CLP. Process SKIP may not be able to terminate immediately since there might be some constraints involving P.END defined in the WHERE clause. The relation *sat* is required to be evaluated before the termination. Relation $sat(E_1, A, T, E_2)$ and $sat(E_1, T, E_2)$ are used to test whether the current state fulfills the requirements. Relation $sat/3$ handles event transition and $sat/2$ handles timed transition. The $sat/3$ and $sat/2$ rules are defines as:

$sat(E1, termination, T)$
$\quad :- get\_process(E1, N), insert(end(N, T), E1, E2), evaluate(E2).$
$sat(E1, A, T)$
$\quad :- get\_process(E1, N), insert(engage(A, N, T), E1, E2), evaluate(E2).$
$sat(E1, T) :- evaluate(E1).$

[2] tpos/7 indicates the relation *tpos* of arity 7, same for sat/3 and sat/4.

The first rule says that whenever there is a *termination* event, the predicate $P.\text{END} = T$ need to be validated in conjunction with all the current requirements in $E1$. Once the resultant predicate is proved to be valid, we append this predicate to the current set of predicates. The second rule is to validate the case when an event is engaged, by adding the predicate $A.\text{ENGAGE}_i = T$ to the environment. The last rule captures the timed transition relation by evaluating the current environment with the current time. Relation $get\_process(E, N)$ is to find the current named process being executed. $evaluate(E)$ is to evaluate the requirements, namely the constraint store.

In the operational semantics, there is a set of composition operators which are more complex. For instance, the rules associated with the semantics of alphabetized parallel composition operator $P_1 \ {}_X||_Y \ P_2$ are as follows.

$$
\begin{array}{l}
tpos(para(P1, P2, X, Y), T, E1, A, para(P3, P2, X, Y), T, E2) \\
\quad :- member(A, X), not(member(A, Y)), \\
\qquad sat(E1, A, T), tpos(P1, T, E1, A, P3, T, E3), \\
\qquad update(E3, engage(A, T), E2). \\
tpos(para(P1, P2, X, Y), T, E1, A, para(P1, P4, X, Y), T, E2) \\
\quad :- member(A, Y), not(member(A, X)), sat(E1, A, T), \\
\qquad tpos(P2, T, E1, A, P4, T, E3), update(E3, engage(A, T), E2). \\
tpos(para(P1, P2, X, Y), T, E1, A, para(P3, P4, X, Y), T, E2) \\
\quad :- member(E, X), member(E, Y), not(E = termination), \\
\qquad sat(E1, A, T), tpos(P1, T, E1, A, P3, T, E3), \\
\qquad tpos(P2, T, E1, A, P4, T, E4), update(E3, E4, E2). \\
tpos(para(P1, P2, X, Y), T, E1, termination, para(P3, P4, X, Y), T, E2) \\
\quad :- sat(E1, termination, T), \\
\qquad tpos(P1, T, E1, termination, P3, T, E3), \\
\qquad tpos(P2, T, E1, termination, P4, T, E4), \\
\qquad update(E3, E4, end(para(P1, P2, X, Y), T), E2). \\
tpos(para(P1, P2, X, Y), T1, E, [], para(P3, P4, X, Y), T2, E) \\
\quad :- tpos(P1, T1, E, [], P3, T2, E), tpos(P2, T1, E, [], P4, T2, E).
\end{array}
$$

Other parallel composition operation, like $|[\,X\,]||$ and $|||$, can be defined as special cases of the alphabetized parallel composition operator straightforwardly. There is a clear one-to-one correspondence between our rules and the operators which are fully defined at [34].

Therefore, the soundness of the encoding can be proved by showing there is a bi-simulation relationship between the transition system interpretation defined in Section 4.2 and ours, and the bi-simulation relationship can be proved easily via a structural induction.

4.4 Verification of Extended Timed CSP

Once we encode the semantics of processes as CLP rules, well-established constraint solvers like CLP($\mathcal{R}$) [55] can be used to reason about those systems. Operational semantics defined in Section 4.2 are all encoded systematically.

4.4.1 Feasibility Checking

After specifying the tasks using extended Timed CSP in CLP($\mathcal{R}$), the very first task is to check whether the tasks are feasible before simulation or reasoning of the system. Feasibility checking is necessary because there might be a conflict among the set of WHERE clauses of a system, which potentially invalidates any proving result. To perform this task, the conjunction of the WHERE predicates and the healthiness conditions are checked.

The output of the feasibility checking is either *yes* if the tasks are feasible or else *no*. In case the tasks are infeasible, i.e., there is no way to satisfy all the constraints, a minimum set of predicates which conflict each other can be generated so as to facilitate users to correct easily. We use the CLP($\mathcal{R}$) predicate *feasibility_checking*(N, S) to fulfill this purpose, where N is the name of the process that is to be checked, and S is the minimum conflict set. If the process N is a feasible process *feasibility_checking*/2 returns false, otherwise the minimum conflict set S is generated and returned. It is performed by a linear scan on a sequence of constraints. We check whether after removing one constraint, the constraints store becomes satisfiable or not. If it does, then this constraint must be important and have to be put back, otherwise it can be discarded. It is an iterative process until a minimum set is found.

$$
\begin{array}{l}
feasibility\_checking(N, S) \\
\quad :- get\_where(N,\ W), min\_cons\_store(W, S). \\
min\_cons\_store(W, S) \\
\quad :- find\_min(1,\ W, S). \\
find\_min(N,\ W,\ W) :- size(W, L), L < N, !. \\
find\_min(N,\ W, S) :- size(X, L), L >= N, \\
\qquad delete\_at(N,\ W,\ WS), (satisfy(WS) ->\ \\
\qquad find\_min(N + 1,\ W, S);\ find_m in(N,\ WS, S)).
\end{array}
$$

Relation $min\_cons\_store(W, S)$ is to generate the minimum conflict set S of W if $W \models false$. It is performed by a linear scan on a sequence of constraints. We check whether after removing one constraint, the constraints store becomes satisfiable or not. If it does, then this constraint must be important and have to be put back, otherwise it can be discarded. It is an iterative process until a minimum set is found.

4.4.2 Reasoning about Safety and Liveness

Feasibility checking is to check whether the tasks modeled in extended Timed CSP are feasible. Once it is proven to be feasible, we can reason about safety or liveness properties by making explicit assertions.

Relation $reachable(P, Q, E1, E2, T1, T2, Tr)$ is defined to explore the full state space if necessary. It states that "process P starts at $T1$ with environment $E1$ and is able to be executed to Q at $T2$ with environment changed to $E2$ via trace Tr".

$$
\begin{array}{l}
reachable(P, P, \_, \_, T, T, []). \\
reachable(P, Q, E1, E2, T1, T2, N) \\
\quad :- tpos(P, T1, E1, A, P1, T3, E3), \\
\qquad (A = t(\_);\ A == tau;\ A = reccall(\_)), \\
\qquad nottable(P1), assert(table(P1)), \\
\qquad reachable(P1, Q, E3, E2, T3, T2, N). \\
reachable(P, Q, E1, E2, T1, T2, [E \mid N]) \\
\qquad :- tpos(P, T1, E1, A, P1, T3, E3), \\
\qquad not(A = t(\_);\ A == tau;\ A = reccall(\_)), \\
\qquad nottable(P1), assert(table(P1)), \\
\qquad reachable(P1, Q, E3, E2, T3, T2, N).
\end{array}
$$

reachable/7[3] is used to build assertions for various property checking. The first property of interest is to find one particular feasible execution for process, provided that the process is feasible. Relation $trace(P, Tr, T)$ is able to generate such feasible trace Tr of process P, whose execution time is T.

$$trace(P, Tr, T) :- notfeasibility_checking(N, \_),$$
$$init\_Env(N, E), reachable(P, \_, E, \_, 0, T, Tr).$$

Reachablity checking is performed by executing "$? - trace(P, Tr, T)$, $property(Tr, Prop)$", which is to find a trace Tr that satisfies some property $Prop$. For example, event a is always engaged before b in Tr.

One property of special interest is deadlock-freeness. Relation $deadlock(P, Tr)$ is used to check the deadlock-freeness property, by trying to find a counterexample where P is deadlocked at some trace Tr.

$$deadlock(P, Tr) :- init_Env(P, E),$$
$$reachable(P, P1, E, E2, 0, T2, Tr),$$
$$(tpos(P1, T2, E2, [t(\_)], Q1, T3, E3) ->$$
$$not(tpos(Q1, T3, E3, A, \_, \_, \_), notA = [t(\_)]);$$
$$not(tpos(P1, T2, E1, A, Q1, T3, \_), notA = [t(\_)])).$$

It states that a process P at time 0 may result in deadlock if it can evolve to the process expression Q at time $T2$ where no event transition is available neither at $T2$ nor at any later moment. The last line outputs the trace which leads to a deadlock. Alternatively, we may present it as the result of the deadlock proving. Note that the above is different from the deadlock checking for standard Timed CSP as presented in [29]. Here the WHERE clauses at each step must be fulfilled. In general, a deadlock-free Timed CSP process may become a non deadlock-free process after it is enriched with certain WHERE clauses. It is, however, also possible for a non deadlock-free process to become deadlock-free.

We can also find the execution duration of a specific event, more specifically, the range of time that the event is able to be engaged. Relation $engage\_time(P, E, R)$ is defined for the

[3] reachable/7 indicates the relation *reachable* of arity 7.

purpose, which is to find the range R of the engage time of event E in process P. The detailed definition for all relations can be found in [34].

$$engage\_time(P, E, []) :- nothappen\_at(P, E, \_).$$
$$engage\_time(P, E, R) :- happen\_at(P, E, T),$$
$$\quad union(R, T, R1), engage\_time(P, E, R1).$$

where R is the range of engaged time of event E in process P which is generated after executing the relation.

4.5 Case Studies

4.5.1 Extended Railway Crossing System

We model a railway control system which is used to control trains passing a critical point such as a bridge. When a train approaches the bridge it sends a signal to the controller within a certain distance. If the bridge is occupied, then the controller sends a stop signal within 10 time units to prevent the train from entering the bridge. Otherwise, if the train does not receive a stop signal within 10 time units, it will start to cross the bridge within 20 time units. The crossing train is assumed to leave the bridge within 3 to 5 time units.

$$\begin{aligned}
Train_i \quad &\widehat{=} approach.i \rightarrow ((stop.i \xrightarrow{7} \\
&\quad start.i \rightarrow entercross.i \xrightarrow{3} leavecross.i \rightarrow Train_i) \\
&\quad \triangle_{10}\ entercross.i \xrightarrow{3} leavecross.i \rightarrow Train_i) \\
&\quad \text{WHERE } entercross.i.\text{ENGAGE} - approach.i.\text{ENGAGE} \leqslant 20, \\
&\qquad leavecross.i.\text{ENGAGE} - entercross.i.\text{ENGAGE} \leqslant 5, \\
&\qquad start.i.\text{ENGAGE} - end.i.\text{ENGAGE} \leqslant 10 \\
Control_{\langle\rangle} \quad &\widehat{=} approach.i \rightarrow Control_{\langle i \rangle} \\
Control_{\langle j \rangle^\frown s} \quad &\widehat{=} approach.i \rightarrow stop.i \rightarrow Control_{\langle j \rangle^\frown s^\frown \langle i \rangle} \ \square \\
&\quad out.j \rightarrow Control_s \ \square \\
&\quad start.j \rightarrow Control_{\langle j \rangle^\frown s} \\
&\quad \text{WHERE } stop.i.\text{ENGAGE} - approach.i.\text{ENGAGE} \leqslant 10 \\
Trains \quad &\widehat{=} Train_1 \ ||| \ Train_2 \ ||| \ Train_3 \\
System \quad &\widehat{=} Trains \parallel Control_{\langle\rangle}
\end{aligned}$$

After transferring this system into CLP($\mathcal{R}$) syntax, we can apply proving over this system. Selected properties are shown as follows:

- Where it is a feasible system.

  Property: feasible(system)

  $?-$ *feasibility_checking*(*system*, S).

  `Feasible`

- Min and Max time for $Train_1$ cross the bridge.

  *Property: Min = min(leavecross.1.*ENGAGE*) , Max= max(leavecross.1.*ENGAGE*)*

  $?-$ *set*(*approach*.1_*engage*, 0), *engage_time*(*system*, *leavecross*.1, R).

  `R>=13, R<=25`

- Whether two trains can cross bridge at the same time

  *Property: -3<= entercross.1.*ENGAGE*-entercross.1.*ENGAGE*<=3*

  $?-$ *happen_at*(*system*, [*entercross*.1, *entercross*.2], [$T1$, $T2$]), $T2 - T1 <= 3$.

  `False`

4.5.2 Real Time Multi-lift System

In this section, we model a real time multi-lift system in Timed Planning specification and hence verify various properties over the system. This real time multi-lift system is initially described and modeled in TCOZ [74]. We also extend this multi-lift system with extra requirements which cannot be modeled using Timed CSP or TCOZ.

The multi-lift system for a building consists of multiple lifts each providing transport between the various floors for the building. The detailed description of the system is introduced in [74].

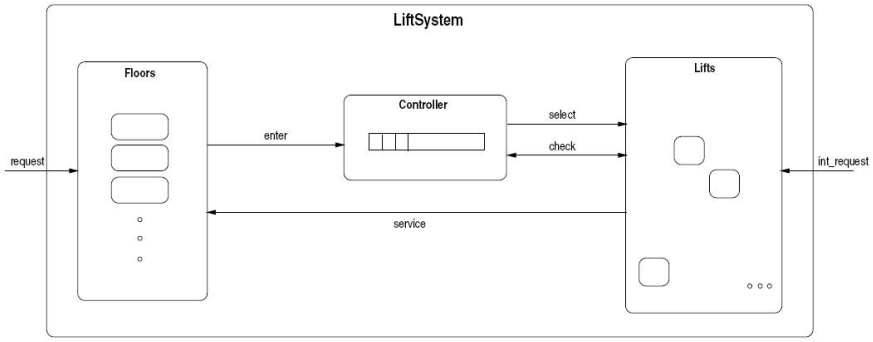

Figure 4.1: The Multi-Lift System Communication Diagram

Each component of the system is modeled as a Timed Planning process, where data information of the system are modeled as global or local variables. Components of the system are communicating with each other through channels. The system consists of several components: Floors, Controller and lifts (Figure 4.1). Inside each lift, there are four parts: a door for allowing access to and from the lift, a shaft for transporting the lift, an internal queue for determining the lift itinerary and a lift controller (Figure 4.2). We model two components *Door* and *Shalf* in this section and the specification of the system is partially shown in Appendix B.

Lift Door

Lift door is modeled as a separate process which can communicate with a servomechanism that activate the door to open or close through a channel *servo* and on another channel *sensor* to determine when the door is open or closed. When the door is closing, a user can press the outside button to request the door for opening again. To be fair to other users in the lift or other users waiting for the lift, it's not possible to keep the door open if one is always pressing the button. So one more constraint is added to the lift door controller

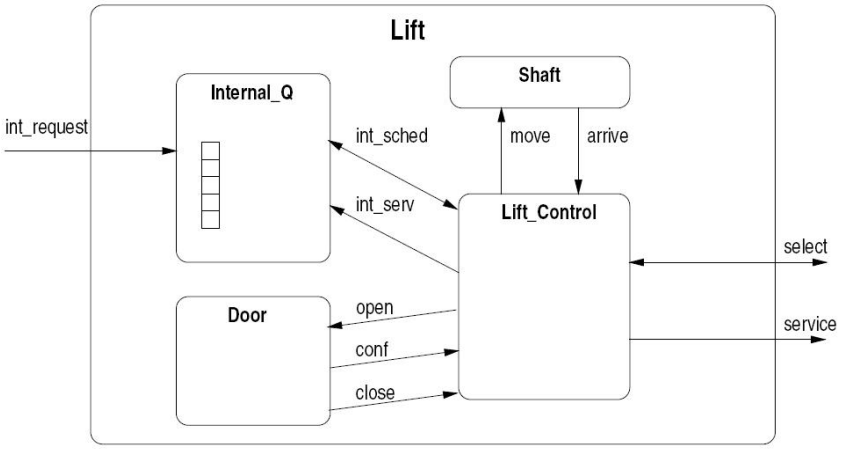

Figure 4.2: Internal Lift Communication Diagram

which would force the door close within t_d after it's opened.

$$
\begin{aligned}
OpenDoor(i) &\mathrel{\widehat{=}} servo!(i, toOpen) \rightarrow sensor?(i, opened) \rightarrow \textsc{Skip} \\
CloseDoor(i) &\mathrel{\widehat{=}} servo!(i, toClose) \rightarrow sensor?(i, closed) \rightarrow \textsc{Skip} \\
CycleDoor(i) &\mathrel{\widehat{=}} OpenDoor(i);\ confirm \rightarrow (\mu \bullet CD\ \textsc{Wait}[t];\ CloseDoor \\
&\quad \triangledown\ sensor?(i, interrupt) \rightarrow OpenDoor;\ CD) \\
Door(i) &\mathrel{\widehat{=}} open?i \rightarrow CycleDoor(i);\ close!i \rightarrow Door(i) \\
&\quad \textsc{Where} close.\textsc{Engage}_i - open.\textsc{Engage}_i \leqslant t_d
\end{aligned}
$$

Shaft

Shaft would receive the number of floors to be moved from channel *move* and then it would move to the destination floor. t is the time to move one floor up or down, *delay* is the acceleration and braking delay. The time for the shaft to reach the destination floor will be in the range of $[t * n,\ t$*n+delay]. The model of shaft is shown as follows where i is the index of this shaft.

$$
\begin{aligned}
Shaft(i) \mathrel{\widehat{=}} & move?n \rightarrow arrive \rightarrow Shaft(i) \\
& \textsc{Where} n * t \leqslant arrive.\textsc{Engage} - move.\textsc{Engage} \\
& \qquad \wedge\ arrive.\textsc{Engage} - move.\textsc{Engage} \leqslant n * t + delay
\end{aligned}
$$

After encoding this system into CLP rules, verification can be applied to the system. Selected properties are shown as follows:

- Is it a feasible system?

  $? - feasibility\_checking(system, S).$

  ```
  Feasible
  ```

- Is the system deadlock free?

  $? - deadlock(system, Tr).$

  ```
  Deadlock free
  ```

4.6 Summary

In this chapter, we presented an extension of Timed CSP to support stating system requirements which constraints all behavioral traces of given processes, for example, *deadline* and execution time of a process, time-related constraints among events which are common requirements for many real time systems and etc. A Timed Planning model is made up of a compositional timed process and a set of constraints over processes, events and the data variables which are the requirements that the process should satisfy. In this proposed formalism, each process is associated with a set of localized timing/untiming requirements with keyword WHERE which can be specified in a compositional way. The full syntax and operational semantics of Timed Planning language are formally defined in this chapter.

A reasoning mechanism for the Timed Planning is hence developed based on Constraint Logic Programming (CLP) by extending the reasoning engine for Timed CSP. Both syntax and semantics of Timed Planning are formally translated into CLP. The operational semantics is encoded to CLP, where a set of global and local variables need to be captured during the execution. Feasibility checking and various property verification can be applied to check systems modeled in Timed Planning. Feasibility checking is necessary which helps users to debug the conflicts of the set of timing constraints specified in the systems.

Chapter 5

Job-shop Scheduling Problems

In many application domains, we are interested in selecting, among all possible behaviors, one that *optimizes* some sophisticated performance measure. We apply Timed Planning and its reasoning engine to solve classical job-shop scheduling problems, where finding an optimal schedule corresponds to finding a shortest execution (in terms of elapsed time). The observation underlying is that classical scheduling and resource allocation problems can be modeled very naturally using Timed Planning whose runs correspond to feasible schedules. In this case, the job-shop scheduling problem can be reduced to a problem of finding a complete execution (an execution that terminates) with the minimum execution time. In our work, Both deterministic [58] and preemptive [80] job-shop scheduling problems are able to be solved. We also apply our approach to handle the extended job-shop scheduling problems, where jobs can have more complex relations, such as a composition of operational behaviors with communications, and jobs with deadlines and relative timing constraints, which no other current work are able to support.

This chapter is organized as follows. In Section 5.1, we first introduce the deterministic job-shop scheduling problem, and then we present how the problems can be modeled and solved using Timed Planning specifications. In Section 5.2, we formalize the preemptive job-shop scheduling problems into Timed Planning and then solve the problem by finding its optimal

scheduler using its underlying reasoning engine. Extended job-shop scheduling problem is shown in Section 5.3. In Section 5.4, a set of experiments are carried out hard bench marks for job-shop scheduling problems. A summary of the work is presented in Section 5.5.

5.1 Deterministic Job-Shop Scheduling Problem

The job-shop scheduling problem (JSSP) is a generic resource allocation problem in which common resources ("machines") are required at various time points (and for given durations) by different jobs. The goal is to find a way to allocate the resources such that all the jobs terminate as soon as possible, which is a schedule with the minimum time interval to finish all jobs. The difficulty is both theoretical (even very constrained versions of the problem are NP-hard) and practical (an instance of the problem with 10 jobs and 10 machines, proposed by Fisher and Thompson [41], remained unsolved for almost 25 years, in spite of the research effort spent on it). The difference between a deterministic and a preemptive job-shop scheduling problem is for the latter case, jobs can use a machine for some time, stop for a while and then resume from where they stopped. We define both problems formally as follows.

5.1.1 Formal Definition of Deterministic Job-shop Scheduling Problem

Definition 5 *(Job-shop scheduling problem) Given a set of $\mathcal{O}$ operations, a set $\mathcal{M}$ of* m *machines, and a set $\mathcal{J}$ of* n *jobs. For each operation $\nu \in \mathcal{O}$ there is a processing time $p(\nu) \in \mathbb{N}$, a unique machine $M(v) \in \mathcal{M}$ on which it requires processing, and a unique job $J(\nu) \in \mathcal{J}$ to which it belongs.*

It has the following requirements:

1. *On $\mathcal{O}$ a binary relation A is defined, which represents the* precedences *of operations: if $(\nu, \omega) \in A$, then ν has to be performed before ω.*

2. *Relation A induces a total ordering of the operations belonging to the same job; no precedence exists between operations of different jobs.*

3. *If* $(\nu, \omega) \in A$ *and there is no* $\upsilon \in \mathcal{O}$ *with* $(\nu, \upsilon) \in A$ *and* $(\upsilon, \omega) \in A$*, then* $M(\nu) \neq M(\omega)$.

□

Definition 6 *(Feasible Schedules for Deterministic Job-Shop Problem) A schedule is a function* $S : \mathcal{O} \to \mathbb{N}$ *that for each operation* ν *defines a start time* $S(\nu)$*. A schedule* S *is* feasible *if*

1. *Covering* :
   $\forall\, \nu \in \mathcal{O} : S(\nu) \geqslant 0,$
2. *Non-Preemption* :
   $\forall\, \nu, \omega \in \mathcal{O}, (\nu, \omega) \in A : S(\nu) + p(\nu) \leqslant S(\omega)$
3. *Mutual-Exclusion* :
   $\forall\, \nu, \omega \in \mathcal{O}, \nu \neq \omega, M(\nu) = M(\omega) :$
   $S(\nu) + p(\nu) \leqslant S(\omega)$ *or* $S(\omega) + p(\omega) \leqslant S(\nu).s$

The problem is to find an optimal *schedule, i.e., a feasible schedule with minimum processing time.* □

Example 5.1.1 Consider M=$\{m_1, m_2\}$ and two jobs $J^1 = (m_2, 4)$ and $J^2 = (m_1, 3), (m_2, 4), (m_3, 6)$. The schedules S_1 and S_2 are depicted in Figure 5.1. The length of S_2 13 is the optimal schedule for a deterministic problem.

end

5.1.2 Modeling with Timed Planning

In this section, we show how deterministic job-shop scheduling problems are modeled in Timed Planning processes. Hence the existing reasoning engine of Timed Planning can be

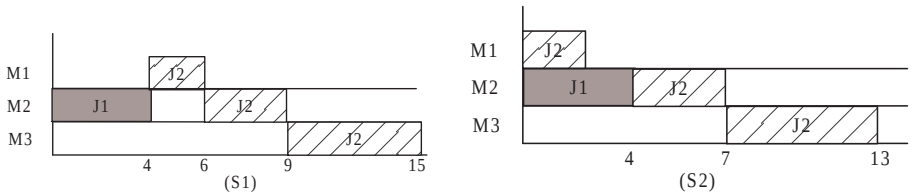

Figure 5.1: The Gantt-Chart representations of two schedules

used to find the optimal schedules.

In our approach, each job is defined as a Timed Planing process and every machine that the job runs on is modeled as an event.

Definition 7 *(Timed Planning for a Job) Let* $J^i = \langle o_1, ..., o_n \rangle$ *be a job, which is a chain of operations (ordered) on a set of machine* $\mathcal{M}$*. Its associated process presentation* P_i *is*

$$P_i \mathrel{\widehat{=}} m_1 \overset{p(o_1)}{\rightarrow} m_2 \overset{p(o_2)}{\rightarrow} ... m_k \overset{p(o_k)}{\rightarrow} \textsc{Skip}$$

where event m_j *in* P_i *denotes operation* o_j *of job* J^i *starting to process on machine* m_j *where* $m_j = \mathcal{M}(o_j)$*.* $p(o_j)$ *is the processing time required for* o_j *on* m_j*. Hence the delay process* $m_j \overset{p(o_j)}{\rightarrow} m_{j+1} \overset{p(o_{j+1})}{\rightarrow} ...$ *denotes that* o_j *must process at machine* m_j *for* $p(o_j)$ *time units; then* o_{j+1} *can start to process which does not need to start immediately.* □

As defined in Definition 6, mutual exclusion is a requirement of a feasible schedule, which is formalized in the following.

Definition 8 *(Mutual Exclusion Constraints) Let* $\mathcal{J} = \{J^1, ..., J^n\}$ *be a job-shop specification and* $\mathcal{P} = \{P_1, ..., P_n\}$ *be a set of timed planning processes defined for each job.*

$$\begin{aligned}
&\textit{mutual-exclusion}(P) \mathrel{\widehat{=}} \\
&\quad \forall\, P_i, P_j \in P, i \neq j, \forall\, m \in \textstyle\sum P_i \cup \sum P_j \bullet \\
&\qquad m \overset{t_i}{\rightarrow} P_i' \preccurlyeq P_i \wedge m \overset{t_j}{\rightarrow} P_j' \preccurlyeq P_j \Rightarrow \\
&\qquad\quad P_i.m.\textsc{Engage} + t_i \leqslant P_j.m.\textsc{Engage} \vee \\
&\qquad\quad P_j.m.\textsc{Engage} + t_j \leqslant P_j.m.\textsc{Engage}
\end{aligned}$$

□

Definition 9 *(Timed Planning for Job-Shop specifications) Let* $\mathcal{J} = \{J^1, ..., J^n\}$ *be a job-shop specification and* $\mathcal{P} = \{P_1, ..., P_n\}$ *be a set of timed planning processes defined for each job. The timed planning presentation for the job-shop specification* $\mathcal{J}$ *will be an interleaving of all processes* P_i *with the non-delay and mutual exclusion constraints:*

$$JSSP \mathrel{\widehat{=}} |||_{0<i\leqslant n} \ P_i \text{ WHERE } \textit{mutual-exclusion}(JSSP)$$

For every complete execution of JSSP (which terminates), its associated schedule S is a feasible *schedule. An optimal schedule is a trace of JSSP with the minimum ending time* $min(JSSP.\text{END})$. □

The associated schedule of every complete execution of *JSSP* is a *feasible* schedule.

- ***Covering***: For each P_i, every execution which terminates guarantees that each and every event in P_i is engaged. *JSSP* which is an interleaving of all P_i also guarantees that all events are engaged at every complete execution.
- ***Non-Preemptive***: Non-Preemptive states that for every ordered pair of operations (ν, ω) of the same job, the later one must start after the full completion of the previous one. The process $m_\nu \overset{p(\nu)}{\rightarrow} m_\omega \overset{p(\omega)}{\rightarrow} ...$ guarantees that operation ω is engaged after $p(\nu)$ time unit of ω.
- **Mutual Exclusion**: This requirement is captured by the *mutual-exclusion*$(JSSP)$ defined in Definition 8, which guarantees that no two jobs evolve the same machine at the same time.

5.1.3 Optimal Schedulers

To find the optimal schedule modeled in Timed Planning, we use a standard depth-first search algorithm to explore the full pathes.

Definition 10 (Non-delay Run) *Let A be the set of enabled events for process P. non-delay constraint says that once process P starts executing, at least one of the enabled event with no side effects must be engaged at the same time.*

$$\begin{aligned} &\mathit{non\text{-}delay} \mathrel{\widehat{=}} \forall\, P \bullet \exists\, e \in \mathit{enable}(P) \wedge e.\textsc{Engage} = P.\textsc{Start} \\ &\mathit{JSSP} \quad \mathrel{\widehat{=}} \quad |||_{0<i\leqslant n} \;\; P_i \\ &\qquad\quad \textsc{Where} \mathit{mutual\text{-}exclusion}(\mathit{JSSP}) \wedge \mathit{non\text{-}delay} \end{aligned}$$

□

The corresponding timed planning representation for the two jobs in Definition 5.1.1 are depicted as follows:

$$\begin{aligned} &P_1 \quad \mathrel{\widehat{=}} m_1 \xrightarrow{4} \textsc{Skip} \\ &P_2 \quad \mathrel{\widehat{=}} m_1 \xrightarrow{3} m_2 \xrightarrow{6} \textsc{Skip} \\ &\mathit{JSSP} \mathrel{\widehat{=}} P_1 \;|||\; P_2 \\ &\qquad \textsc{Where}(P_1.m_1.\textsc{Engage} + 4 \leqslant P_2.m_1.\textsc{Engage} \vee \\ &\qquad\qquad P_2.m_1.\textsc{Engage} + 2 \leqslant P_1.m_1.\textsc{Engage}) \end{aligned}$$

As illustrated in the above section, finding an optimal schedule is to find a execution of timed planning process *JSSP* with the minimum ending time, i.e., *min(JSSP.*End*)*. In the rest part of this section, we apply several approaches to reduce the size of the transition system and arrive at a more heuristic search algorithm.

Max Non-delay Test

Partial order reduction is a technique for reducing the size of the state-space to be searched by a model checking algorithm. It exploits the commutativity of concurrently executed transitions, which result in the same state when executed in different orders. For example:

$$\begin{aligned} &P_1 \quad \mathrel{\widehat{=}} m_1 \xrightarrow{4} \textsc{Skip} \\ &P_2 \quad \mathrel{\widehat{=}} m_2 \xrightarrow{5} \textsc{Skip} \\ &\mathit{JSSP} \mathrel{\widehat{=}} P_1 \;|||\; P_2 \\ &\qquad \textsc{Where} \mathit{non\text{-}delay} \wedge \mathit{mutual\text{-}exclusion}(\mathit{JSSP}) \\ &\qquad \equiv m_1 \to m_2 \xrightarrow{5} \textsc{Skip} \sqcap m_2 \to m_1 \xrightarrow{5} \textsc{Skip} \\ &\qquad\quad \textsc{Where} \mathit{non\text{-}delay} \end{aligned}$$

The *JSSP* process is a choice of two process, where the order of the events denotes decision to whom to give the first resource. The order of m_1 and m_2 does not affect the result. Hence, *JSSP* is equivalent to $m_1 \rightarrow m_2 \stackrel{5}{\rightarrow}$ SKIP WHERE*non-delay*. The non-delay predicate in the process guarantees that once m_1 is engaged, m_2 must be engaged immediately, which is m_1.ENGAGE= m_2.ENGAGE. The job-shop process *JSSP* can be further simplified to "$JSSP \mathrel{\widehat{=}} \{m_1, m_2\} \stackrel{5}{\rightarrow}$ SKIP", where $\{m_1, m_2\}$ is a new notation, which indicates that event m_1 and m_2 are engaged simultaneously.

Consider the case that n jobs whose first operations are running on m machines where $m<n$ which means there are more or one jobs whose first operations are running on the same machine. It cannot directly apply the partial order reduction rules which still need to expand the transition system.

Definition 11 (Max Non-delay Run) *Let A be the set of events for process P that are enabled. Max non-delay constraint says that once process P starts executing, at least one subset of A with maximum distinct events must be engaged at the same time.*

$$max\text{-}non\text{-}delay \mathrel{\widehat{=}}$$
$$\forall P, \exists E \in max\_sub(P) \bullet E.\text{ENGAGE} = P.\text{START}$$

where enable(P) *is the set of enabled events of* P*,* $max\_sub(P)$ *is a set of max subset of enable*(P) *with non duplicate events and no side effects, which means that no pair of events in* $max\_sub(P)$ *evolving the same* m_i *and does not preserves any order implicitly.* □

Never Better Test

The max non-delay test removes the redundant paths where order of events evaluates to be the same. Since we are interested only in optimal executions, we can apply stronger reductions that do not preserve all runs, but still preserve the optimal runs. During the execution, we always keep the latest most optimal schedule, *shortest(M)*. Whenever we reach a

subprocess, we do a never better test, by estimating the best length of each subprocess. By comparing the best length of the subprocess with the *shortest(M)*, if the recent subprocess can never result a better schedule, which means even this subprocess can reach its best length, it won't be shorter than *shortest(M)*, then we will not expand this subprocess.

Best First Test

The searching algorithms described above all require the full state space search. The next improvement consists of using a more intelligent search. At each step, a set of $max\_sub(jssp)$, maximum subset of enabled events with no duplicates, are enabled. We define a pre-evaluation ranking function $Order(jssp)$ for ordering the $max\_sub(jssp)$ of enabled events. We define another evaluation function $Est(jssp)$ for estimating the possible best length of the process *jssp*.

Algorithm 1 Best First Test

procedure *BestFirst_test(jssp)*

```
Optimal := ∞
if jssp == stop then
  Optimal:= 0;
else
  if Est(jssp) < Optimal then
    Sets:= Order(max_sub(jssp))
    for all s such that s ∈ Sets do
      p' := next(jssp, s, T)
      Optimal:= min{Optimal, BestFirst_test(p')+T}
    end for
  end if
end if
return Optimal;
```

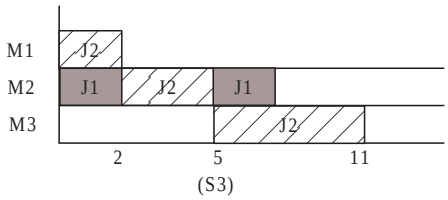

Figure 5.2: The Gantt-Chart representations of a preemptive schedules

5.2 Preemptive Job-Shop Scheduling Problem

We extend the results on deterministic Job-shop scheduling problem to preemptive jobs, i.e. jobs that can use a machine for some time, stop for a while and then resume from where they stopped.

Example 5.2.1 Recall Example 5.1.1, two feasible schedule for the two jobs $J^1 = (m_2, 4)$ and $J^2 = (m_1, 3), (m_2, 4), (m_3, 6)$ in shown in Figure 5.1. S_2 is the optimal schedule if it is a deterministic job-shop scheduling problem. Consider it is a preemptive problem, a feasible schedule S_3 is generated. In S_3, J^1 is preempted at time $= 2$ on machine m_3 to give the machine to J^2. After J^2 has finished its operation on m_3, J^1 resumes its previous operation on m_3. The length of S_3 is 11 and it is a optimal schedule. **end**

The definition of a job-shop specification (Definition 5 remains the same. The main difference between deterministic and preemptive job-shop scheduling problem is the feasible schedules where the latter one does not satisfy the $Non - Preemption$ requirement in Definition 6.

Definition 12 *(Feasible Schedules for Preemptive Job-Shop Problem) Let $T(\mathcal{O}, i) \in \mathbb{N}$ be processing time of the i_{th} step at which operation $\mathcal{O}$ executes. A schedule is a relation $S \subseteq \mathcal{O} \times \mathbb{N} \times T$ so that (ν, st, t) $\in S$ indicates that operation ν starts to process on time st and processes for time t. A schedule S is* feasible *if*

1.*Ordering* :
$\forall \nu, \omega \in \mathcal{O}, (\nu, \omega) \in A,$
$(\nu, st, t) \in S, (\omega, st', t') \in S : st + t \leqslant st' + t'$
2.*Covering* :
$\forall \nu \in \mathcal{O} : \sum_{(\nu,st,t)\in S} t = p(\nu)$
3.*Mutual-Exclusion* :
$\forall \nu, \omega \in \mathcal{O}, \nu \neq \omega, (\nu, st, t) \in S, (\omega, st', t') \in S,$
$M(\nu) = M(\omega) : st + t \leqslant st' \ or \ st' + t' \leqslant st.$

□

5.2.1 Solving Preemptive Job-shop Scheduling Problems

Definition 13 *(Timed Planning for a Preemptive Job) Let* $J^i = \langle o_1, ..., o_n \rangle$ *be a job, which is a chain of operations (ordered) on a set of machine* $\mathcal{M}$*. Each operation* o_i *should process on machine* m_j *for* $p(o_i)$ *time unit. Its associated process presentation for* o_i *is:*

$$O_i \mathrel{\widehat{=}} \mu X \bullet m_s \rightarrow m_e \rightarrow \textsc{Skip};\ X \ \Box \ \textsc{Skip}$$
$$\textsc{Where} \sum m_e.\textsc{Engage} - m_s.\textsc{Engage} \leqslant p(o_i) \wedge$$
$$\sum m_e.\textsc{Engage}\text{-}m_e.\textsc{Engage} = p(o_i) \Leftrightarrow \textsc{End} < \infty$$

Event m_s *denotes that operation* o_i *starts to process on its corresponding machine* m*,* m_e *denotes that it leaves* m*. Expression* $\sum m_e.\textsc{Engage} - m_s.\textsc{Engage}$ *represents the total time* o_i *processes on* m*.*
The associated process presentation P_i *for each job is* $P_i \mathrel{\widehat{=}} O_{i_1};\ O_{i_2};\ ...;\ O_{i_k}$*.* □

Definition 14 *(Mutual Exclusion Constraints) Let* $\mathcal{J} = \{J^1, ..., J^n\}$ *be a job-shop specification and* $\mathcal{P} = \{P_1, ..., P_n\}$ *be a set of timed planning processes defined for each job.*

$$mutual\text{-}exclusion(P) \mathrel{\widehat{=}}$$
$$\forall P_i, P_j \in P, i \neq j, \forall \{m_s, m_e\} \subseteq \sum P_i \cap \sum P_j \bullet$$
$$(\mu X \bullet m_s \rightarrow m_e \rightarrow ... \preccurlyeq P_i \wedge$$
$$\mu X \bullet m_s \rightarrow m_e \rightarrow ... \preccurlyeq P_j) \Rightarrow$$
$$P_i.m_e.\textsc{Engage} \leqslant P_j.m_s.\textsc{Engage} \vee$$
$$P_j.m_e.\textsc{Engage} \leqslant P_i.m_s.\textsc{Engage}$$

□

Definition 15 *(Timed Planning for Preemptive Job-Shop specifications) Let* $\mathcal{J} = \{J^1, ..., J^n\}$ *be a job-shop specification and* $\mathcal{P} = \{P_1, ..., P_n\}$ *be a set of timed planning processes defined for each job. The timed planning presentation for the preemptive job-shop specification* $\mathcal{J}$ *will be an interleaving of all processes* P_i *with mutual exclusion constraints:*

$$PJSSP \mathrel{\widehat{=}} |||_{0<i\leqslant n} \ P_i$$
$$\text{WHERE} \mathit{mutual\text{-}exclusion}(PJSSP)$$

□

For every complete execution of *PJSSP* (which terminates), its associated schedule S is a *feasible* schedule. An optimal schedule is a trace of *PJSSP* with the minimum ending time $min(JSSP.\text{END})$. The solution of this problem turns out to be finding the minimum execution time of process *PJSSP*.

The associated schedule of every complete execution of *PJSSP* is a *feasible* schedule, which satisfies the three requirements defined in 12:

- ***Covering***: For each P_i, every execution which terminates guarantees that each and every event in P_i is engaged. *JSSP* which is an interleaving of all P_i also guarantees that all events are engaged at every complete execution.
- ***Covering***: Non-Preemptive generally says that for every ordered pair of operations (ν, ω) of the same job, the later one must start after the full completion of the previous one. The process $m_\nu \overset{p(\nu)}{\rightarrow} m_\omega \overset{p(\omega)}{\rightarrow}$ ... guarantees that operation ω is engaged after $p(\nu)$ time unit of ω.
- **Mutual Exclusion**: This requirement is captured by the mutual-exclusion(PJSSP) defined in Definition 8, which guarantees that no two jobs evolving the same machine

at the same time.

5.3 Extended Job-shop Scheduling

Timed Planning is flexible to specify system behaviors with operational behaviors and critical timing constraints. Therefore, those extended job shop scheduling problems with several additional features that are often specified in task scheduling problems can be easily and naturally modeled and solved using Timed Planning. In this section, we focus on the compositional behaviors and the deadline and relative times extension. To the best of our knowledge, no other approaches are capable for those extensions.

Compositional Job Behaviors

In traditional job-shop scheduling problems, all jobs are executed synchronously, which means that they are enabled at the same time. In our approach, jobs can be constructed in a more general way, for example, we can specify choices of jobs, sequences of jobs and interruption of one job by another job, by using the composition operators in Timed Planning. For example, a job-shop scheduling problem consists of 4 jobs $\mathcal{J} = \{J^1, J^2, J^3, J^4\}$, where either J^1 or J^2 is running concurrently with J^3 which is interrupted by J^4 in 10 time units after execution. This scheduling problem can be specified as follows:

$$JSSP \mathrel{\widehat{=}} (P1 \,\Box\, P2) \,|||\, (P3 \,\triangledown_{10}\, P4)$$
$$\text{WHERE} \mathit{mutual\text{-}exclusion}(JSSP) \wedge \mathit{non\text{-}delay}$$

In our interpretation, jobs can communicate with each other through channels. A job $P1$ can activate another job $P2$ after some time t or after $P1$ finishes its first i_{th} operations. A job can also stop another job for some time units.

An extended job-shop scheduling problem consists of 2 jobs J^1, J^2, $J^1 = (1,2), (0,3), (2,5)$, $J^2 = (2,3), (0,3), (1,3)$. J^1 will activate J^2 after J^1 finishes its first operation. The Timed

Planning model of this problem is specified as follows where a channel naming c is used to fulfill this purpose.

$$
\begin{array}{ll}
P1 & \widehat{=} 1 \xrightarrow{2} c!activate \rightarrow 0 \xrightarrow{3} 2 \xrightarrow{5} \textsc{Skip} \\
P2 & \widehat{=} c?activate \rightarrow 2 \xrightarrow{3} 0 \xrightarrow{3} 1 \xrightarrow{3} \textsc{Skip} \\
JSSP & \widehat{=} P1 \,|||\, P2 \\
 & \quad \textsc{Where} mutual\text{-}exclusion(JSSP) \wedge non\text{-}delay
\end{array}
$$

Deadlines and Relative Timing Constraints

Another extension of job-shop scheduling problem is that we can specify the deadline, relative timing constraints of each job. Those constraints can be naturally handled in Timed Planning by using its *Where* predicate. We formulate the constraints into 3 rules as follows:

Rule 1 Every operation o_i must be imperatively terminate before time $d(o_i)$

$$\forall\, o_i \in \mathcal{O} \bullet S(o_i) + p(o_i) \leqslant d(o_i)$$

Rule 2 Every job J^i must terminate before time $d(J^i)$

$$\forall\, J^i \in \mathcal{J} \bullet J^i.\textsc{End} \leqslant d(J^i)$$

Rule 3 Relative timing constraints between operations

$$\exists\, o_i, o_j \in \mathcal{O} \bullet p(o_i) \odot p(o_j) < t, where \odot \in \{+, -\}$$

5.4 Experiments

In order to show the efficiency of the optimization, we carry out an experiment for n jobs with 4 operations, n = 2,...,6[1]. We compare performance in terms of the state space and execution time of algorithms employing, progressively, the max non-delay and the cut test. The obtained results are depicted in Table 5.1 (m.o. indicates memory overflow), where $n = 2, ..., 6$. $\sharp j, \sharp states, time$ represents number of jobs, number of state space and the

[1]The problems can be found in `http://www.comp.nus.edu.sg/~zhangxi5/horae/jobshop1.txt`

Problem Size			Max non-delay		Never Better Test	
♯j	♯states	time(s)	♯states	time(s)	♯states	time(s)
2	764	0.07	16	0.01	12	0
3	6475	0.44	43	0.02	33	0
4	1.07e+06	40.8	202	0.04	96	0.01
5	m.o.	m.o.	4813	0.5	584	0.2
6	m.o.	m.o.	112299	5.9	4614	1.2

Table 5.1: The result for n jobs with 4 machines, where $n = 2, ..., 6$. $\sharp j, \sharp states, time$ represents number of jobs, number of state space and the execution time.

execution time. The table shows the performance, in terms of execution time (in seconds) and state space, of algorithms employing, progressively, none, max non-delay test and the never better test. As we can see from Table 5.1, Never better test has much better performance in turns of both state space and execution time.

We test ten problems among the benchmarks of the job-shop scheduling problems on Windows XP with a 2.0 GHz Intel CPU and 2 GB memory. In Table 5.2, we compare our best result on the problems with the result reported in Table 15 of the survey paper [58], as well as the best result among randomly generated solutions for each problem. The first three columns give the problem name, no. of jobs and no. of machines. Out results (time in seconds, length of the best schedule and the deviation) appear next. The following two columns shows the best out of 2000 randomly-generated solutions, followed by the optimal result of each problem.

5.5 Summary

In this chapter, we presented a novel approach of solving classic job-shop scheduling problem using Timed Planning and underlying reasoning engine. We applied Timed Planning to solve

Problem			Timed Planning			Random		Opt
name	♯j	♯m	time(s)	length	deviation	length	deviation	length
FT10	10	10	18	1001	7.63%	1761	89.35%	930
LA02	5	10	2	720	9.90%	1056	61.68%	655
LA19	10	10	0.2	902	7.12 %	1612	91.45%	842
LA21	15	10	102	1104	5.54%	2339	123.61%	1046
LA24	15	10	66	1007	7.58%	2100	124.00%	936
LA25	15	10	19	1098	12.38%	2209	126.10%	977
LA27	20	10	25	1441	16.68%	2809	127.45%	1235
LA29	20	10	112	1357	17.79%	2713	135.50%	1152
LA36	15	15	35	1341	5.57%	2967	133.90%	1268
LA37	15	15	56	1489	6.58%	3188	128.20%	1397

Table 5.2: The result for the ten hard bench marks deterministic job-shop scheduling problems. The first three columns give the problem name, no. of jobs and no. of machines

both deterministic and preemptive job-shop scheduling problems. There are some work using formal method approach to solve job-shop scheduling problems. [2] proposed a mechanism for modeling the job-shop scheduling problems in Timed Automata and hence finding the optimal solutions. [3] presented an approach of solving preemptive job-shop scheduling problem using stopwatch Timed Automata where some of the clocks might be freezed at certain states.

In our approach, the job-shop scheduling problem can be naturally modeled as Timed Planning processes. We also worked with the extended job-shop scheduling problems, where all jobs have composition operational behaviors. Besides, jobs with deadline and relative timing constrains are also able to be captured in our approach. We believe that the insight gained from this point of view will contribute both to scheduling and to the study of timed processes. We have demonstrated that the performance of the Timed Planning approach of solving job-shop scheduling problem can be highly improved by applying a set of optimizations. There are still many potential improvements to be explored to reduce the execution time, such as new partial-order methods and heuristics, etc.

Chapter 6

Modeling and Verification of Timed Security Protocols

Security protocols are widely used for securing application-level data transport crossing distributed systems, typically by exchanging messages constructed using cryptographic operations (e.g. message encryption). In general, designing security protocols is notoriously difficult and error-prone. Many protocols proposed in the literature and many protocols exploited in practice turned out to be awed, or their well-functioning was found to be based on implicit assumptions. Since the late eighties various approaches [22, 46, 19, 15, 20, 16] have been put forward for the formal verification of security protocols to overcome the problems of faulty implementations and hidden requirements.

The new challenges are raise when different timing aspects are required in the security protocol design, such as timestamps, delays, timeout and a set of timing constraints. In the past years, there has been an increasing interest in the formal analysis of timed cryptographic protocols. However, there are few tool supports for modeling and analyzing security protocols with the capability of capturing various timing features. A particularly successful approach to analyze untimed security protocols is using CSP [51] to model and CSP model checker FDR [42] to analyze protocols [87, 70]. Motivated by this approach, we focus

on timed extensions of CSP to accomplish the modeling and analyzing of timed security protocols, particularly Timed Planning.

In this chapter, we show how to model timed security protocols by using Timed Planning language proposed in this thesis. Our approach is different from the previous approaches by taking the timing information into account. The use of explicit timing information allows us to specify security protocols with timestamps, timeout and retransmissions which can be naturally modeled using the specification. In the timing analysis, we could verify timed non-injective agreement authentication property which can be easily extended to other authentication property verification [46]. We also propose a novel approach of using the capability of Timed Planning to avoid such attacks without changing the original specifications of the protocols. Besides, we can model timing requirements/constraints and verify other timed sensitive properties such as execution time of a protocol which is beyond the capability of existing approaches.

6.1 Formal Specification of Timed Security Protocols

In this section, we show how to model timed security protocols using Timed Planning in a structured way. All the protocols we consider here have a similar objective: in each protocol, an *initiator* A seeks to establish a session with a *responder* B, possibly with the help of a *sever* S, where A, B, and S are *principal*s.

*Principal*s (including initiator, responder and server) and intruder are modeled as Time Planning processes. The whole network would be the parallel execution of all processes. Event $send.S.R.M$ is introduced to denote the behavior sending the message M from sender S to receiver R. Event $receive.S.R.M$ denotes receiver R receives a message M from sender S.

We use the Wide Mouth Frog protocol (WMF) [13] as a running example to illustrate the ideas. The Wide Mouth Frog protocol is a computer network authentication protocol

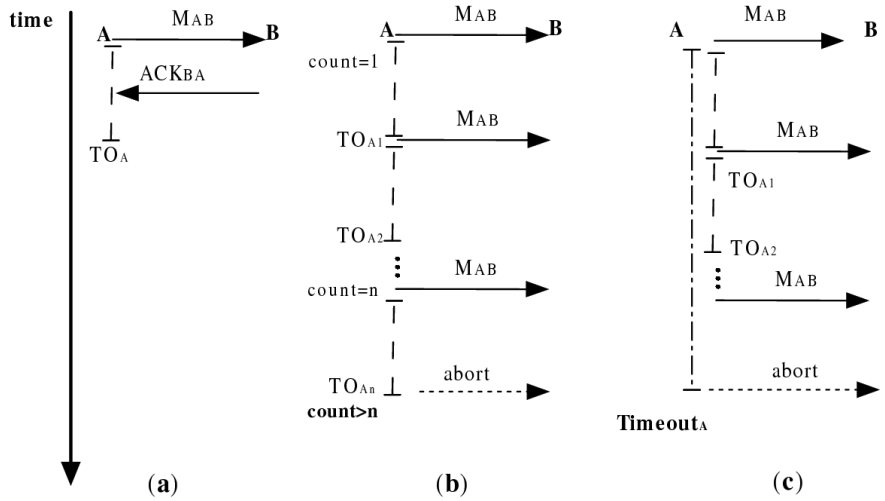

Figure 6.1: Timeout patterns: (a) typical (b) count-bounded (c) time-bounded

designed for insecure networks. The goal is to allow two principals A and B to exchange a secret key Kab via a trusted server S. The model is described as follows, where Kas and Kbs are shared keys of A and B with sever S respectively. Ta and Ts are timestamps generated and sent by A and S respectively.

A, B, S : *principal*
Kas, Kbs, Kab : *Key*
Ta, Ts : *timestamp*
1. $A \rightarrow S$: $A, \{Ta, B, Kab\}Kas$
2. $S \rightarrow B$: $\{Ts, A, Kab\}Kbs$

Timeout and Retransmissions

Look at the following simple protocol, where A sends a message M_{AB} to B, and later B will send an acknowledgement ACK_{BA} to A after receiving M_{AB}.

$1.A \rightarrow B : M_{AB}$
$2.B \rightarrow A : ACK_{BA}$

After principal A sends a message in a session of a protocol, A starts a timer that will *timeout* if A does not get ACK_{BA} from the receiver B. When a timeout is reached, the principal A

can execute two actions: retransmit or reset a session. We are able to model both actions, where an implementation of the protocol should specify which action to perform. In some study with the introduction of timeout [16], principal A will resend the message if it detects a timeout. But they do not discuss the case if the resent message also gets timeout. In our specification, we introduce a bounded timeout, including count-bounded timeout and time-bounded timeout discussed as follows.

- *Count-bounded timeout* After principal A sends a message in a session of a protocol, it will start a timer and a counter. Once detecting a timeout, it will resend this message. If the resent message also gets timeout, it would resend again, by increasing number of resending by 1. If the number of times exceeds the max value, A will send request to abort this protocol (see Figure 6.1 (b)).

- *Time-bounded timeout* After principal A sends a message in a session of a protocol, it starts two timers. If it detects a timeout from the first timer, it will resend this message; if the resent message also gets timeout, A will resend again. Once the second timer reaches a timeout for the whole sending message process, A will send a request to abort this protocol (see Figure 6.1(c)).

Timed Planning specification of both count-bounded timeout and time-bounded timeout are shown in process $CBTimeout(d, max)$ and $TBTimeout(d_1, d_2)$, respectively, where d and d_1 are the timeout for sending a message, max is the maximum retransmission times and d_2 is the timeout for the whole process. $c = \text{TES} \downarrow send.A.B.\{m_{AB}.A\}$ is the number of times of retransmission.

$$
\begin{aligned}
&CBTimeout(d, max) \mathrel{\widehat{=}} (\mu\, X \bullet send.A.B.\{m_{AB}.B\} \\
&\qquad \rightarrow (receive.B.A.\{ack_{BA}.B\} \rightarrow \text{SKIP} \overset{d}{\triangleright} X)) \\
&\qquad \triangledown\ ([c > max]\text{SKIP}) \\
&\qquad \text{WHERE} c = \text{TES} \downarrow send.A.B.\{m_{AB}.A\} \\
&TBTimeout(d_1, d_2) \mathrel{\widehat{=}} (\mu\, X \bullet send.A.B.\{m_{AB}.B\} \rightarrow \\
&\qquad (receive.B.A.\{ack_{BA}.B\} \rightarrow \text{SKIP} \triangledown_{d_1} X)) \\
&\qquad \overset{d_2}{\triangleright} \text{SKIP}
\end{aligned}
$$

Timestamps and Lifetime

Timestamp is a typical way to prevent replay attacks, by simply attaching the current time value to a message. It is later used by the receiver of this message to make sure that it was recently generated, not a replay. A timestamp of a message can be easily recorded by $m.Engage$ which is the engage time of the event m, where event m denotes sending the message. In our implementation, we keep TES, a set of all timed events, which is a record of all engage time of all event engaged so far. It is easy to check whether $m.Engage$ is the most recent one or not by using the predefined predicate $fresh(m, m.Engage, TES)$ in Section 4.1.

Initiator, Responder and Server

We model each principal (initiator, responder, server) as a process. For the Wide Mouth Frog protocol, the behavior of the initiator A is sending a message $\{T_a, b, k_{ab}\}$ to Server S using public key k_{as} and waiting for acknowledgement from S, where t_a is the timestamp. The responder B receives message from the server and then send the acknowledgment to S. The server receives message from A and then message $\{T_s, a, k_{ab}\}$ to B with new timestamp T_s. The three components are modeled as follows.

$$
\begin{array}{ll}
Initiator & \widehat{=}\ start\_encrpt \rightarrow end\_encrpyt \rightarrow \\
 & \quad (\mu\, X \bullet send.A.S.\{T_a.b.k_{ab}\}_{k_{as}} \\
 & \quad (receive.S.A.\{ack_{SA}\} \rightarrow \textsc{Skip} \overset{d_1}{\triangleright} X)) \\
 & \quad \triangledown_{d_2} \textsc{Skip} \quad \textsc{Where} \\
 & \quad T_a = send.A.S.\{T_a.b.k_{ab}\}_{k_{as}}.\textsc{Engage}_1 \\
Responder & \widehat{=}\ receive.S.B.\{T_s.a.k_{ab}\}_{k_{bs}} \rightarrow \\
 & \quad send.B.S.\{ack_{SB}\} \rightarrow start\_decrypt \rightarrow \\
 & \quad end\_decrypt \rightarrow \textsc{Skip} \quad \textsc{Where} \\
 & \quad T_s \leqslant receive.S.B.\{T_s.a.k_{ab}\}_{k_{bs}}.\textsc{Engage} \\
 & \quad \wedge\ fresh(receive.S.B.\{T_s.a.k_{ab}\}, T_s, \textsc{Tes})^1
\end{array}
$$

[1] fresh(e,t,Tes) is defined in Section 4.1.

$$
\begin{aligned}
\mathit{Server} \;\widehat{=}\; & \mathit{receive}.A.S.\{\,T_a.b.k_{ab}\}_{k_{as}} \rightarrow \\
& \mathit{send}.S.A.\{\mathit{ack}_{SA}\} \rightarrow \mathit{start\_decrypt} \rightarrow \\
& \mathit{end\_decrypt} \rightarrow (\mu\, X \bullet \\
& \mathit{send}.S.B.\{\,T_s.a.k_{ab}\}_{k_{bs}} \rightarrow \\
& (\mathit{receive}.B.S.\{\mathit{ack}_{bs}\} \rightarrow \textsc{Skip} \overset{d_3}{\triangleright} X)) \\
& \triangledown_{d_4} \textsc{Skip} \quad \textsc{Where} \\
& T_s = \mathit{send}.S.B.\{\,T_s.k_{ab}\}_{k_{bs}}.\textsc{Engage}_1 \\
& \wedge\, \mathit{fresh}(\mathit{receive}.A.S.\{\,T_a.b.k_{ab}\},\, T_a,\, \textsc{Tes}) \\
\mathit{Network} \;\widehat{=}\; & \mathit{Initiator} \;||\; \mathit{Responder} \\
& ||\; \mathit{Server} \;||\; \mathit{Cryptograph}
\end{aligned}
$$

The network is a parallel composition of the three processes, as well as the *Cryptograph*.

Intruder

The intruder works basically as a Dolev-Yao intruder [22]. The difference is that our intruder takes time. The intruder can impersonate each agent executing the protocol, so it can play each of the roles in the protocol. Even thought the intruder has got its own keys, nonces etc., it can also try to use all the information it is receiving in the protocol run as its own (e.g., nonces). For the purpose of this paper, we restrict the behaviors of the intruder that it cannot read the mind of other principals to get some secret and it is unable to guess values. We restrict the behaviors of the intruder to the following actions:

- encrypt and decrypt a message
- intercept a message
- replay a message
- send a message to any principals
- delay a message with arbitrary time

The models of intruder and the new system with intruder are specified as follows:

$$
\begin{aligned}
Intruder \mathrel{\widehat{=}} & \ start\_encrypt \rightarrow end\_encrypt \rightarrow Intruder \\
& \Box\ start\_decrypt \rightarrow end\_decrypt \rightarrow Intruder \\
& \Box\ receive.S.I.M \rightarrow Intruder \\
& \Box\ send.I.R.M \rightarrow Intruder \\
& \Box\ receive.S.I.M \xrightarrow{T} send.I.R.M \rightarrow Intruder \\
System \ \ \mathrel{\widehat{=}} & \ Network \parallel Intruder
\end{aligned}
$$

where I is the identity of itself, S is the sender, R is the receiver and M is the message. The intruder would intercept into the protocol sessions through channel *send* and *receive*.

We present a natural way for specifying security protocols using Timed Planning, including the initiator, responder, server and intruder, with timestamps and timeout. It shows that Timed Planning is a good mechanism to model timed security protocols in a compositional way.

6.2 Verification of Authentication

For verifying timed security protocols, protocols are firstly modeled in Timed Planning models, and then translated into CLP programs. Properties which need to be verified are encoded into CLP goals using relations defined in Section 4.4. In this section, we show how to define and verify timed security properties, including timed authentication properties. Moreover, by using timing information of each protocol run, potential attacks are also to be found.

6.2.1 Timed Authentication Property

Authentication property is very important in security protocols. Protocols need to accomplish authentication of the *Initiator* and *Responder*. [46] classified a set of authentication requirements. We will focus on timed non-injective agreement, which is an extension of the non-injective agreement defined in [46].

Definition 16 (Timed Non-Injective Agreement) *A timed security protocol guarantees* timed non-injective agreement*, only if responder B thinks it has completed a run of the protocol with A using data D, then A was actually running the protocol with B using data D. To check the authentication property, signals are added to principals to indicate principal B has completed a protocol run with A and A is actually running a protocol with B, whenever necessary.* □

Our method is to insert signals, which are special kind of events, into each process, and then to check the corresponding relationship of these signals. In our approach, event $commit.B.A.D$ is used to denote that principal B has completed a protocol running with A using data D, $run.A.B.D$ as principal A is running a protocol with B using data M.

$$
\begin{aligned}
Initiator \mathrel{\widehat{=}} & \ start\_encrpt \rightarrow end\_encrpyt \rightarrow \\
& run.a.b.\{t_a.b.k_{ab}\} \rightarrow \\
& (\mu\, X \bullet send.A.S.\{t_a.b.k_{ab}\}_{k_{as}} \\
& \rightarrow (receive.s.a.\{ack_{sa}\} \rightarrow \textsc{Skip} \overset{d_1}{\triangleright} X)) \\
& \triangledown_{d_2} \textsc{Skip} \\
& \textsc{Where} t_a = send.a.s.\{t_a.b.k_{ab}\}_{k_{as}}.\textsc{Engage}
\end{aligned}
$$

Properties which need to be verified are specified as assertions. There is a one-to-one relationship between the runs of A and B for protocol satisfying timed non-injective agreement property. The timed non-injective agreement means that once there is a *commit* event, there should be at least one *run* event appearing previously. The assertion is defined as:

$$
\begin{aligned}
& non\_inj\_agr(P, Tr) :- trace(P, Tr, \_), \\
& \qquad end(Tr, commit.B.A.M), not\ in(Tr, run.B.A.M).
\end{aligned}
$$

This predicate is to find a trace Tr, where there is no event *run* occurred before event *commit*. Tr is an instance of an attack. Predicate $trace(P, Tr, T)$ is defined in previous section. After checking the timed non-injective agreement property over WMF protocol, such attack has been found where responder B finds that it has more than one sessions with initiator A but in fact there should be only one [6]. We also model the Needham-Schroeder Public-Key protocol as process NSP, by executing the following goal:

$$? - non\_inj\_agr(NSP, Tr).$$

We are able to find an attack where the responder B commits a session with the initiator A, but A did not establish a protocol run with B, where A is running the protocol with the intruder I [70]. The trace is:

$\langle start\_encrypt, end\_encrypt, send.A.I.M,$
$receive.A.I.M, send.I.B.M, receive.I.B.M,$
$send.B.I.M2, receive.B.I.M2, send.I.A.M2,$
$receive.I.A.M2, send.A.I.M3, receive.A.I.M3,$
$send.I.B.M4, receive.I.B.M4, commit.B.A.D\rangle$

Preventing Attacks

By preventing the authentication attack, we propose appending a constraint WHERETES $\downarrow$ $run.A.B.D \geqslant$ TES $\downarrow$ $commit.B.A.D$ to the protocol process to guarantee number of event *commit* is always less than event *run*. The revised Wide Mouth Frog protocol is modeled as WMF_R1.

Another approach of preventing attacks is by changing the timeout values. There is a particular timed authentication attack for Wide Mouth Frog protocol where the intruder can extend the life time of a (possibly compromised) key Kab as wanted, whereas A and B think that it has expired and been destroyed [87].

$i.1.$ $A \rightarrow S$: – A, {Ta, B, Kab}Kas
$i.2.$ $S \rightarrow I(B)$: – {Ts, A, Kab}Kbs
$ii.1.$ $I(B) \rightarrow S$: – B, {Ts, A, Kab}Kbs
$ii.2.$ $S \rightarrow A$: – {T's, B, Kab}Kas
$iii.1.$ $I(A) \rightarrow S$: – A, {T's, B, Kab}Kas
$iii.2.$ $S \rightarrow B$: – {T"s, A, Kab}Kbs

This attack cannot be checked using timed non-injection agreement because there is exactly one session for both initiator and responder. For step $i.2$ I takes the place of B to get the message from server S where S should be expecting an *ack* from B. If S does not get the *ack* within the timeout window, it could choose to terminate the session. Let d_n be the network transaction delay for each message passing. S should be able to get the *ack* message from B after $2 * d_n$ if there is no attack. By changing the timeout window for S to less than

Property	Protocol	Description	Result
P1	CoreWMF	non-injection agreement	Yes
P2	CoreWMF	timed non-injection agreement	Yes
P3	CoreWMF + Intruder	timed non-injection agreement	No
P4	CoreWMF + Intruder	replay attack	Yes
P5	CoreWMF + Intruder	timed authentication attack	Yes
P6	WMF_R1 + Intruder	timed non-injection agreement	Yes
P7	WMF_R1 + Intruder	replay attack	No
P8	WMF_R1 + Intruder	timed authentication attack	Yes
P9	WMF_R2 + Intruder	timed non-injection agreement	Yes
P10	WMF_R2 + Intruder	replay attack	No
P11	WMF_R2 + Intruder	timed authentication attack	No

Figure 6.2: Analysis of Wide Mouth Frog protocols

$6 * d_n$, this attack will be avoided because B can only send *ack* to S after step *iii*.2, which takes more than $6 * d_n$ time unites. We model the Wide Mouth Frog protocol with bounded timeouts in *WMF_R2*.

We list a set of properties over Wide Mouth Frog protocols, where the results are shown in Figure 6.2. *CoreWMF* is the protocol without intruder; and *CoreWMF + Intruder* is the protocol with intruder I. The results are computed in minutes or even seconds in PC with 2.83GHz Intel Q9550 CPU and 2 GB memory. Comparison with other approaches are ignored since there is no other tool supporting timed analysis of security protocols.

6.2.2 Using Timing Information

We are able to check other timing properties of the protocol. For example, find the minimum and maximum execution time of a run of a protocol P by finding P.End, using predicate *engage_time*(P, *termination*, R) defined in Section 4.4.

$$execution\_time(P, R) :- engage\_time(P, termination, R).$$

where *termination* is a special event denoting the end of the execution, R is the range of execution time of protocol P.

By using the minimum and maximum execution of a protocol run, we can also check *timing authentication* properties. In our approach, if there is a run of a protocol between two principals A and B, it must be finished within a time interval $[T_{min}, T_{max}]$, without intruders. If a protocol run ends before T_{min}, this may be a result of an attack which omits at least one instructions or performs at least one instructions faster than it is expected. If a protocol run ends after T_{max}, this may be a result of an attack performing with extra actions, such as replays. We define a predicate *time_attack* (P, Tr, Min, Max) to find a trace which exceeds the time interval $[min, max]$, which is also an instance of an attack.

$$time\_attack(P, Tr, Min, Max) :- \\ trace(P, Tr, T), (T < Min;\ T > Max).$$

Assume for a protocol run, once a sender sends a message to the receiver, the message reaches the receiver within [2, 4] time units because of the network delay. for WMF protocol, without introducing an intruder, the execution time of a protocol run is [4,8]. By executing goal $?-$ *time_attack* $(WMF, Tr, 4, 8)$. We find a trace whose execution time is more than 8. If the execution time of a protocol run exceeds the expected time interval, there must be some attack in this protocol run. But if the execution time is within the time interval, attack-free is not guaranteed.

6.3 Summary

In this work, we have proposed a new method of modeling and analyzing timed security protocols which consists of various timing aspects such as timestamps, delays, timeouts and a set of timing constraints. To fulfill the aim, we substantially Timed Planning with capabilities to stating complicated and critical timing requirements of timed security protocols,

in a compositional way. Based on our previous work on building a reasoning tool for Timed CSP, a prototype mechanized proving system, based on CLP($\mathcal{R}$) to verify various properties over systems modeled in this extended specification has been built. We model principals as processes, as well as the cryptograph device, including timestamps, timeout, retransmissions and delays. The timed non-injective agreement authentication property can be verified using our underlying reasoning engine, which can be easily extended to verify other authentication properties. We propose a novel approach to find timing attacks using timing information of protocol sessions. We can also model timing requirements of the protocols in our WHEREpredicates and verify other timing properties of the protocols.

There are many works on analyzing security protocols. In literature, methods for formal verification of security protocols do not take time into account, and this choice simplifies the analysis [8]. Powerful theorem provers like Isabelle and PVS have been applied to verify timed dependent security properties [7, 39]. A common practice in the area of modeling and verification of security protocols is to abstract away timestamps [9]. Our approach is similar to [87] which uses CSP to model and analyze untimed security protocols. Recently there have been also other approaches to verify such protocols [20], which does not discuss timeout and retransmissions. One more recent work, using Timed Automata for verifying timed security [16], also introduces timeout. The difference of our timeout and retransmissions is that we propose a bounded timeout, which also consider timeout over retransmissions. Moreover, they cannot model timedstamps which needs global synchronization on clocks.

As a future work, we will apply Timed Planning specifications to other domains, such as timed scheduling problems, complex real-time systems and etc. For analyzing timed security protocols, we will expand the verification of security properties, such as secrecy, integrity, fairness and the timing properties such as the time range for an easy attack. For our underlying reasoning engine, there are many potential improvements to be explored to reduce the execution time, such as symmetry reduction and heuristics.

Chapter 7

Modeling and Verification of Pervasive Computing

Timed Planning is a powerful modeling language. In this section, we demonstrate the effectiveness of this language by applying it into pervasive computing domain.

Alzheimerąś disease is characterized by progressive deterioration of the patientąś intellectual capacities that evolves during a period of 7 to 10 years on the average [59]. In most developed countries, the demographical, structural and social trends tend towards more and more elderly people, which will create dramatic impact on the society on both financial and organizational aspects. Dementia is a progressive, disabling, chronic disease affecting 5% of all persons above 65 years old and over 40% of people above 90 [43, 44]. Elders with dementia often have declining short-term memory and have difficulties to remember necessary activities of daily living (ADLs) [38]. It is widely believed that the reminding system is a potential way to assist elderly people to complete ADLs [35]. From the demographic changes, we can expect a rise in the quantity of aging people with mild dementia. The impact of such increment will be long waiting lists for sheltered housing projects, homes for the elderly, nursing homes and other care facilities. It has been suggested that most aging people prefer to stay at home as long as possible, even if they are at risk [14]. From

social, economic and elder's perspectives, it is very important to enable the elders with mild dementia to stay in their own homes safely and regularly as long as possible and to reduce the burden of care givers. Even though, it is still a great pressure on nursing homes and other similar type of care facilities. It also increases the pressure on care givers [12].

Pervasive computing techniques have been proposed to assist elders with mild dementia to improve their level of independence and quality of life through cognitive reinforcement. To support formal analysis, we propose an approach of using Timed Planning to model and verify critical properties. In this chapter, we model and verify systems in two different settings. Firstly, in Section 7.1, a context-aware reminding framework for elders living at home alone is built and modeled using Timed Planning. Secondly, in Section 7.2, a reminding framework for elders living at nursing home is modeled using Timed Planning. The first case study focuses on the various reminders and conflicts between reminders. The second case study focuses on the various sensors and reasoning about the rules.

7.1 Timed Reminding System for Dementia Elderly at Home

In smart home systems [109, 36], a centered design approach is adopted with user studied at three different sites [76, 38]: Amsterdam (Netherland), Belfast (UK) and Lulea (Sweden). The summary of needs and wants from the three workshops is illustrated in Table 7.1.

As can be seen from the user study, a well designed reminding service is a fundamental function for assisting elders with mild dementia living at home. The reminding system can help them to perform daily activities properly and regularly, and also increase their feeling of safety and security at home.

[109] identifies the activities needed to be reminded to the elders are: meal preparation, brushing teeth, making phone calls, appointments, turning off stoves, closing refrigerators, taking medication, bringing the keys and locking doors when going outside. All of those activities are modeled using Timed Planning and presented later, where we also consider

Area of Cognitive Reinforcement	**Summary of required services/solutions for people with mild dementia**
Remembering	Item locator (keys, mobile devices)
Maintaining Social Contact	Means to provide communication support with carer/family network
Performing daily activities	Support with daily activities associated with pleasure Control of household devices e.g., television, radio, planning activities
Enhanced feeling of safety	*Warnings of doors left open* Warning for devices left on General: way to contact others in instances of emergency

Table 7.1: Summary of needs and wants for mild dementia people living at home

some other activities such as flushing and washing hands after toilet, turning off lights/tv when going to sleep.

7.1.1 Reminding Service Classifications

Based on the observations of elder's daily activities described above, [36] classifies the reminding services into four kinds of categories, based on time and event. Time-based services are used to capture the daily schedule of the elder people based on calendar time. However, the dynamic nature of people's daily activities poses potential risks for the elderly where prompting services based on activities are necessary.

Time-based Prompting Two kinds of prompting service relate to time are described below.

- Timed fixed prompting services: the prompting is issued at pre-defined time such that the subject understands the urgency of executing certain activity.
- Timed relevant prompting services: the prompting is relevant to a time, but can be delayed within an acceptable time window.

Event-based Prompting Two kinds of prompting service triggered by events are described below.

- Event urgent prompting services: triggered by events and need to be prompt immediately.
- Event related prompting services: triggered by events, but can be delayed up to an acceptable delay time, or must be delayed after an min_delay time.

Timed fixed promoting services are reminders be executed at certain time, such as time for wake up, time for appointments, etc. Timed relevant prompting services are the reminders that are more flexible than time fixed reminders, such as preparing lunch between 11:30am to 12:30pm. Event urgent prompting services are those reminders must be prompted immediately when some event is performed, for example, stove must be turned off if the elderly is going to leave the house. Event related prompting services are reminders triggered by events but can be delayed within an acceptable time window or must be delayed to some time, for example, medication must be taken after 30 minutes of lunch.

7.1.2 Modeling using Timed Planning

Modeling time with calendar time

One important issue for timed reminding system is that how to model calendar time using Timed Planning specification. As we all know that Timed CSP only considers relative time where all time are relative to the start time of the process.

There are two possible options to model the calendar time.

1. Abstract the detailed information, use a 'channel' to get the information required from the environment.

2. Build extra functions to compute the current calendar time using the starting time of the system and the relative current time to it. Most programming languages have the library to support those functions.

In this work, we choose the second option, which uses extra techniques to model the calender time in terms of (week, hour, minute and second). Every time the system starts, record the start time of the system in the 4 terms. We can easily compute the current calendar time with the start time and relative passing time.

$$
\begin{array}{ll}
Week & : [0..6] \\
Hour & : [0..23] \\
Minute & : [0..59] \\
Second & : [0..59] \\
Calendar & : \langle Week, Hour, Minute, Second \rangle \\
Calendar.week & = Calendar(1) \\
Calendar.hour & = Calendar(2) \\
Calendar.minute & = Calendar(3) \\
Calendar.second & = Calendar(4) \\
Calendar.time & = (Calendar(2), Calendar(3), Calendar(4))
\end{array}
$$

In the following, $calendar\_time(c1, t)$ defines a function which returns the current time according to the start time of the system; $relative\_time(c1, c2)$ defines a function which returns the current time according to the current relative time (in seconds).

$$convert(i, j) = \{(s, t) \mid \forall\, i, j, s, t : \mathbb{R} \bullet t = i \ mod\ j \wedge j * s + t = i\}$$

$$
\begin{array}{l}
calendar\_time(c1, t) = \\
\{c2 \mid \forall\, c1, c2 : Calender, w1, w2 : W, h1, h2 : H, m1, m2 : M, s1, s2 : S, t : \mathbb{R} \bullet \\
\quad (w1, h1, m1, s1) = c1 \wedge (w2, h2, m2, s2) = c2 \Rightarrow \\
\quad\quad \exists\, x, y, z, d : \mathbb{R} \bullet \\
\quad\quad\quad convert(s1 + t, 60) = (x, s2) \wedge convert(m1 + x, 60) = (y, m2) \wedge \\
\quad\quad\quad convert(h1 + y, 24) = (z, h2) \wedge convert(w1 + z, 7) = (d, w2)\}
\end{array}
$$

$$
\begin{aligned}
&\mathit{relative\_time}(c1, c2) = \\
&\{t \mid \forall\, c1, c2 : \mathit{Calendar}, w1, w2 : W, h1, h2 : H, m1, m2 : M, s1, s2 : S, t : \mathbb{R} \bullet \\
&\quad (w1, h1, m1, s1) = c1 \land (w2, h2, m2, s2) = c2 \Rightarrow \\
&\quad\quad \exists\, x, y, z, d : \mathbb{R} \bullet \\
&\quad\quad\quad \mathit{convert}(s1 + t, 60) = (x, s2) \land \mathit{convert}(m1 + x, 60) = (y, m2) \land \\
&\quad\quad\quad \mathit{convert}(h1 + y, 24) = (z, h2) \land \mathit{convert}(w1 + z, 7) = (d, w2)\}
\end{aligned}
$$

$\mathit{convert}(i, j)$ is a function converting number i to a pair (s, t), where s is the multiple of j and t is the mod of j. The purpose of this function is to convert seconds into minutes, minutes into hours and hours into days. For example, converting 63 minutes (1 hours and 3 minutes) would use function $\mathit{convert}(63, 60)$, returning pair $(1, 3)$.

$\mathit{calendar\_time}(c1, t)$ is a function converting the relative t time units passing starting on initial calendar time $c1$ to the current calendar time $c2$. For example, the starting calendar time $c1 = (0, 0, 0, 0)$ (denoting time 0:00:00 am, Sunday), after 90061 seconds passing, the current calendar time would be $c2 = (1, 1, 1, 1)$ (denoting time 1:1:1 am, Monday).

$\mathit{relative\_time}(c1, c2)$ is a function find the relative time between calendar time $c1$ and $c2$, providing $c2$ is later than $c1$.

The following defines a set of global variables:

$c : \mathit{Calendar}$

Modeling location and activities

In this reminding system, locations of the participant are modeled as global variables which can be of type *boolean, integer, string or tuple.* We abstract the changes of the variables which should be updated by the environment. The activities of the participant, which should be detected by sensors, are modeled as *events*, such as *getup*, *leaving*, *take_medication*, *pick_phone* and etc.

Timed fixed reminders (TFR)

Timed fixed reminders: the prompting is issued strongly at defined time so that the subject understands the urgency of executing certain activity. e.g.,

- get up at 7 a.m.
- catch flight at 11:30 a.m.
- attend lecture/meeting/seminar at 2p.m.

Timed fixed reminders can be modeled as processes whose starting time and prompting time must be exactly as the required time.

$$\begin{aligned} TFR(t) \mathrel{\widehat{=}} & [guard]prompt \rightarrow \textsc{Skip} \\ & \textsc{Where}\ calendar\_time(c, \textsc{Start}).time = t \wedge \\ & prompt.\textsc{Engage} = \textsc{Start} \end{aligned}$$

Example 7.1.1 The reminder which wakes up the elder at 7 a.m. would be:

$$\begin{aligned} wake\_up((7,0,0)) \mathrel{\widehat{=}} & [athome \wedge sleeping]prompt \rightarrow \textsc{Skip} \\ & \textsc{Where}\ calendar\_time(c, \textsc{Start}).time = (7,0,0) \wedge \\ & prompt.\textsc{Engage} = \textsc{Start} \end{aligned}$$

end

Timed related reminders (TRR)

Timed related reminders: prompting is related to a time, but can be delayed within an acceptable time. e.g.,

- Start to cook dinner between 5 p.m and 5:30 p.m.
- Start to preparing tutorials/lectures at 3p.m.

At a specific time t, this reminder is enabled, but it does not need to prompt immediately if there are other reminders prompting. It needs to be prompted within an acceptable delay d.

$$\begin{aligned} TRR \mathrel{\widehat{=}} & [guard]prompt \rightarrow \textsc{Skip} \\ & \textsc{Where}\ calendar\_time(c, \textsc{Start}) \geqslant t \wedge \\ & \quad calendar\_time(c, promp.\textsc{Engage}) \leqslant t + d \end{aligned}$$

Example 7.1.2 Remind the elderly to start preparing his/her dinner at around 5:30 p.m. to 5:30 p.m. every day if he/she is at home.

$$\begin{aligned} prep\_dinner((17,0,0),(17,30,0)) \mathrel{\widehat{=}} & [athome \wedge \neg sleeping]prompt \rightarrow \textsc{Skip} \\ & \textsc{Where}\ calendar\_time(c, \textsc{Start}).time \geqslant (17,0,0) \\ & \wedge\ prompt.\textsc{Engage} \leqslant (17,30,0) \end{aligned}$$

end

Event urgent reminders (EUR)

Event urgent reminders: reminders triggered by event, which must be prompted as soon as the event is engaged.

- turnoff the stove after cooking
- bring the key while going out

Event urgent reminders can be modeled as a process whose first event is the triggering event. To make sure the *prompt* event happen immediately, the constraints $trigger\_event.\textsc{Engage} = prompt.\textsc{Engage}$ is added in the Where clause.

$$\begin{aligned} EUR \mathrel{\widehat{=}} & trigger\_event \rightarrow [guard]prompt \rightarrow \textsc{Skip} \\ & \textsc{Where}\ trigger\_event.\textsc{Engage} = prompt.\textsc{Engage} \end{aligned}$$

Example 7.1.3 Remind the elderly to turnoff the stove after he finishes cooking.

$$\begin{aligned} turnoff\_stove \mathrel{\widehat{=}} & cooking\_end \rightarrow prompt \rightarrow \textsc{Skip} \\ & \textsc{Where}\ cooking\_end.\textsc{Engage} = prompt.\textsc{Engage} \end{aligned}$$

end

Event related reminders (ERR)

Event related reminders: prompting is triggered by an event, but can be delayed within an accepted time. For example,

- wash hands after toilet within 3 minutes.
- take medication after 30 minutes of lunch.

We call the minimum delay time after the triggering event mdt, and the maximum delay time after after the triggering event dt. Event related reminders can be modeled as a process whose first event is the triggering event. After that a delay of mdt occurs. The WHERE clause makes sure that the engagement time of *prompt* event is bounded by the maximum delay dt.

$$ERR \mathrel{\widehat{=}} trigger\_event \overset{mdt}{\rightarrow} [guard]prompt \rightarrow \textsc{Skip}$$
$$\textsc{Where}\ prompt.\textsc{Engage} \leqslant trigger\_event.\textsc{Engage} + dt$$

Example 7.1.4 Remind the elderly to wash hand after toilet in 3 minutes if there is no phone call.

$$Wash\_hand \mathrel{\widehat{=}} finish\_toilet \rightarrow [\neg phonecall]prompt \rightarrow \textsc{Skip}$$
$$\textsc{Where}\ prompt.\textsc{Engage} \leqslant finish\_toilet.\textsc{Engage} + 3_{minute}$$

end

Daily/ Weekly reminders

These reminders will prompt at a specific time every day, or every week with some conditions. For example, the wake up reminder which will prompt 8 a.m. every weekdays. The attending lecture reminder which will prompt every 10 a.m. every Monday if not public holidays.

We need other techniques (rules) to identify that whether today is weekday or weekend or public holiday. We have two alternative options:

- retrieve the information of whether today is weekday or weekend from the environment by channels and abstract the details.
- use *Calendar Logic* to calculate those information in our system. Therefore, Ontology can be a good candidate for modeling Calendar Logic.

Daily promoting services: prompt at time t everyday.

$$\begin{aligned} \mathit{Daily\_Reminder} \mathrel{\widehat{=}} & \mathit{Reminder};\ \mathit{Wait}\ 1_{\mathit{day}};\ \mathit{Daily\_Reminder} \\ & \textsc{Where}\ \mathit{calendar\_time}(c, \mathit{Reminder}.\textsc{Start}).\mathit{time} = t \end{aligned}$$

Daily promoting services: prompt at time t everyday, except Saturday and Sunday

$$\begin{aligned} \mathit{Daily\_Reminder} \mathrel{\widehat{=}} & \mathit{Reminder};\ \mathit{Wait}\ 1_{\mathit{day}};\ \mathit{Daily\_Reminder} \\ & \textsc{Where}\ 1 \leqslant \mathit{calendar\_time}(c, \mathit{Reminder}.\textsc{Start}).\mathit{week} \leqslant 5 \\ & \quad \wedge\ \mathit{calendar\_time}(c, \mathit{Reminder}.\textsc{Start}).\mathit{time} = t \end{aligned}$$

7.1.3 Medication Planner

In this subsection, we use a medication planner [104] to demonstrate the modeling of reminders using timed planning. The system requirements of the medication planner are described below.

1. Never prompt outside the window: within a certain time interval: *[st, st+dt]*
2. Do not prompt if pill is already taken within the current window
3. Do not prompt if the participant is not at home: *athome=true*
4. Do not prompt if the participant is sleeping
5. Do not prompt if participant is on the phone
6. Resume prompting if the participant returns home before the window expires
7. Prompt at plan 2 if participant is leaving

8. Wait till the time the user usually takes the pill. If it is earlier than the recommended pill taking time, start checking for plan 1 prompting opportunities at the usual pill time.

9. If only less than 20 minutes left till the window expires, start prompting at plan 1 disregarding all other rules (except 1-3)

There are two kinds of promptings in the system.

Plan 1 Prompt using the nearest device. The chime is played 10 seconds each time and lights stay on till location changes. Stop if pill is taken. Escalate to Plan 2 after 10 minutes.

Plan 2 Prompt using all prompting devices in the house every minute. Lights on devices stay on and chime is played for 10 seconds every minute.

$$Plan1 \mathrel{\widehat{=}} prompt \overset{10}{\rightarrow} \textsc{Skip};\ Wait(10_{minute});\ Plan2 \triangledown taken?yes \rightarrow \textsc{Skip}$$
$$Plan2 \mathrel{\widehat{=}} prompt \overset{10}{\rightarrow} \textsc{Skip};\ Wait(1_{minute});\ Plan2 \triangledown taken?yes \rightarrow \textsc{Skip}$$

$$\begin{array}{lll} Medi \mathrel{\widehat{=}} ([athome \wedge \neg taken \wedge \neg onThePhone \wedge \neg sleeping]Plan1 & & \text{– rule 2 - 6} \\ \quad \Box\ \ leaving?yes \rightarrow Plan2) & & \text{– rule 7} \\ \quad \triangledown_{dt-20_{minute}}\ Plan2 & & \text{– rule 8} \\ \quad \textsc{Where}\ calendar\_time(c, \textsc{Start}) \geqslant st \wedge & & \\ \qquad calendar\_time(c, promp.\textsc{Engage}) \leqslant st + dt & & \text{– rule 1} \end{array}$$

For rule 6: prompting will resume if the participant returns home before the window expires. Our modeling is able to preserve this requirement in a more clever way. Once the participant returns home, the boolean variable *athome* will be changed to *true*, hence this process is able to be executed. So this model satisfies a more general requirement:

- Prompting will resume if the participant returns home or wakes up or finishes phone call before the window expires.

7.1.4 Specific domain: Elderly Reminding System

In this section, we present the Timed Planning model for the elderly reminding systems. The reminders required in the system are listed as follows.

- wake up reminder (Timed fixed reminder)
- brush teeth (Timed related reminder)
- meal preparation (Timed related reminder)
- making phone calls (Timed related reminder)
- appointment (Timed related reminder)
- turn off stoves (Event urgent reminder)
- taking medication (Timed related reminder)
- bring keys while going out (Event urgent reminder)
- wash hands after toilet (Event related reminder)

There are four possible events, which would trigger new reminders, i.e., *get up*, *toilet*, *going out* and *go to toilet*. There are several external events, which cannot trigger new reminders, i.e., *phone call*, *knock the door* and *come back*. The Elderly can choose to either response or not. In the following, we define a set of global variables to keep the status of the elderly.

athome : *boolean*
sleep : *boolean*
goingout : *boolean*
bringkey : *boolean*
washhand : *boolean*

There is one assumption about the system, i.e., the time stamp 00:00 a.m. is set to be 0. Hence 1:00 a.m. is 60, 2:00 a.m. is 120, and so on. In the following, we introduce the reminder models in the system.

Wake up reminder

Wakeup reminder service is provided to wake up dementia patients at a specific time.

The template defined for wakeup reminder is shown as:

$$\begin{aligned} Wakeup(t) \mathrel{\widehat{=}} & [athome \wedge sleeping]prompt\_wakeup \rightarrow \textsc{Skip} \\ & \textsc{Where Start} = calender\_time(c, \textsc{Start}) = t \\ & \quad \wedge prompt\_wakeup = \textsc{Start} \end{aligned}$$

The wakeup reminder set at 7:00 am every morning is specified using process $Wakeup((7,0,0))$, where (7,0,0) represents time 7:00 am.

Watch TV

Reminders for the elderly to watch his favorite TV program at 10am, which is timed fixed reminder.

$$\begin{aligned} TV \mathrel{\widehat{=}} & [athome]prompt\_tv \rightarrow \textsc{Skip} \\ & \textsc{Where } calender\_time(c, \textsc{Start}) = (10,0,0) \wedge prompt\_tv.\textsc{Engage} = \textsc{Start} \end{aligned}$$

Bring key

Bringkey reminder service is provided to remind the elderly to bring his keys while going out, which is an event urgent reminder.

channel:{bring_key} is a sensor detecting whether the elderly brings his keys or not.

channel :{goingout}

$$\begin{aligned} Key \mathrel{\widehat{=}} & goingout?yes \rightarrow ([\neg bringkey]prompt\_bringkey \rightarrow \textsc{Skip} \;\Box\; [bringkey]\textsc{Skip}) \\ & \textsc{Where } prompt\_bringkey.\textsc{Engage} = goingout?yes.\textsc{Engage} \end{aligned}$$

Wash hands

WashHands reminder service is provided to remind the elderly to wash hands after toilet, if the elderly does not wash his hand in one minute. The maximum delay allowed in this

reminder is 5 minutes. This is an event related reminder, which is modeled below.
channel:{toilet, wash_hands}

$$\begin{aligned} \mathit{WashHands} \mathrel{\widehat{=}} \mathit{toilet?finish} \rightarrow (\mathit{wash\_hand?yes} \rightarrow \textsc{Skip} \mathbin{\triangledown_{1_{minute}}} \\ \mathit{prompt\_washhand} \rightarrow \textsc{Skip}) \\ \textsc{Where}\ \mathit{prompt\_washhand}.\textsc{Engage} - \mathit{toilet?finish}.\textsc{Engage} \leqslant 5_{minute} \end{aligned}$$

Prepare Breakfast

WashHands reminder service is provided to remind the elderly to start preparing his breakfast at around 8:30 a.m. to 9:00 a.m. every day if he/she is at home.

$$\begin{aligned} \mathit{PrepareBreakfast} \mathrel{\widehat{=}} [\mathit{athome} \wedge \neg \mathit{sleep}](\mathit{prompt\_breakfast} \overset{45_{minute}}{\rightarrow} \textsc{Skip}) \\ \textsc{Where}\ \mathit{calender\_time}(c, \textsc{Start}) = (8, 30, 0) \wedge \\ \mathit{prompt\_breakfast}.\textsc{Engage} \leqslant \textsc{Start} + 30 \end{aligned}$$

7.2 Smart Nursing Home for Mild Dementia Elderly

The primary goal of this project is to enable the person with mild dementia, through ambient intelligence and assistive technologies, to maximize his physical and mental functions and to continue to engage in social networks, so that he can lead an independent and purposeful life. The person with mild dementia is usually able carry out the actual task in most basic activities of daily living but is often handicapped by his poor memory and thus forgets to carry out these tasks. These can include bathing, changing clothes and taking medication on time. Ambient intelligence enables the capture of context information in a pervasive and non-intrusive manner. Assistive technology that provides timely prompts and reminders will enable him to preserve his abilities and independence.

This project focuses on the automated recognition of activities and behaviors in smart nursing homes and providing assistance / intervention accordingly. We will carry out the automated monitoring of basic Activities of Daily Living (bADL) and instrumental Activities of Daily Living (iADL) among single and multiple residents in smart nursing homes.

Technically, these objectives translate to significant advances in sensitivity and specificity in activity and plan recognition of finer grained bADLs / iADLs for single subject; and improved location tracking, object / human dissociation and activity recognition among multiple subjects.

As mentioned above, the target is to support ageing people with mild dementia in performing Activities of Daily Living (ADLs) at home. Elders with dementia often have declining short-term memory and have difficulties in planning and performing the necessary activities of daily living. It is conceivable that a system of timely reminders is a potential way to assist elderly people to complete bADLs/iADLs, which moderates their memory impairment and reduces the burden of care givers.

7.2.1 System Design

Activity Daily Living (ADLs)

Activities of Daily Living (ADLs) is a term used in health care to refer to daily self-care activities within an individual's place of residence, in outdoor environments, or both. Health professionals routinely refer to the ability or inability to perform ADLs as a measurement of the functional status of a person, particularly in regards to people with disabilities and the elderly. Activity of Daily Living (ADL) monitoring is important in order to determine the well being of elderly persons in their home settings.

In the current phase of this project, we mainly consider ADLs within bed room and shower room which are important activities for early dementias patients living in the nursing home. ADLs are including:

- Enter/Leave bed room
- Sitting/Sleeping on bed
- Go to the shower room for toilet

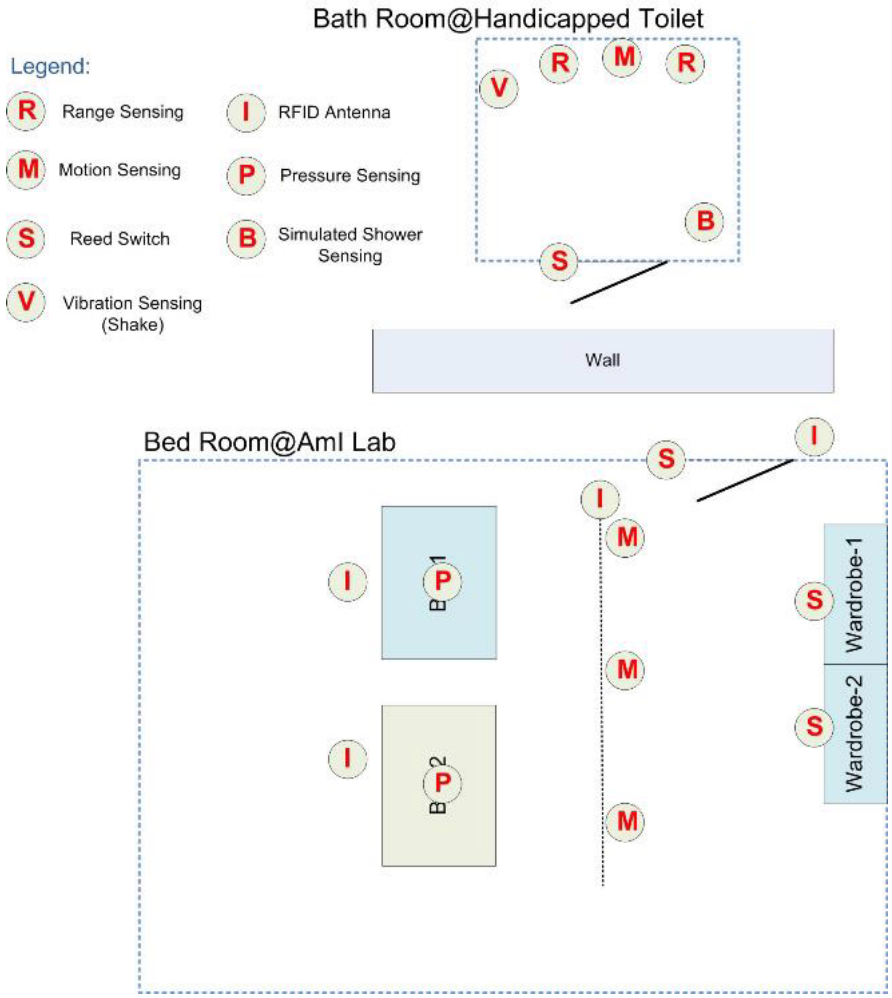

Figure 7.1: The Sensor Setup for Bedroom and Bath Room

- Go to the shower room to wash hands
- Go to the shower room to take shower

Process $Residence(i, status)$ is defined below to model all the possible activities of a residence, where i is the identity of the patient and *status* is the current location of the residence. Initially, each residence is outside of the room, $status = outside$. Activities happened in room are composed of sub-activities, where each sub-activity is modeled as a process. *EnterRoom* specifies the activity entering the bed room if and only if the residence is outside of the bedroom. *LeaveRoom* specifies the activity leaving the bed room if and only if the residence is inside the bedroom. *BedActivity* is defined the capture the activities in bed, which can be *sitting on the bed*, *sleeping on the bed* and *get up*. Detailed models of the three activities are defined as follows, where the complete model of all residence activities is shown in Appendix C. Possible *status* of each residence: {outside, inRoom, inShowerRoom, inBed}

$$
\begin{aligned}
Residence(i, status) &\mathrel{\widehat{=}} (EnterRoom(i, status);\ Residence(i, inRoom)) \\
&\quad \Box\ (LeaveRoom(i, status);\ Residence(i, outside)) \\
&\quad \Box\ (Shower(i, status);\ Residence(i, status)) \\
&\quad \Box\ (BedActivity(i, status);\ Residence(i, status)) \\
&\quad \Box\ (Toilet(i, status);\ Residence(i, status)) \\
EnterRoom(i, status) &\mathrel{\widehat{=}} [status == outside] enterRoom.i \rightarrow \textsc{Skip} \\
LeaveRoom(i, status)) &\mathrel{\widehat{=}} [status == inRoom] leaveRoom.i \rightarrow \textsc{Skip} \\
BedActivity(i, status) &\mathrel{\widehat{=}} [status == inRoom] gotoBed \rightarrow \\
&\quad ((sitonbed.0.i \rightarrow BedAct\_(i, 0)) \\
&\quad \Box\ (sitonbed.1.i \rightarrow BedAct\_(i, 1))) \\
BedAct\_(i, j) &\mathrel{\widehat{=}} (SittingOnBed(i) \Box Sleeping(i)); \\
&\quad (BedAct_(i, j) \Box GetUp(i, j)) \\
SittingOnBed(i) &\mathrel{\widehat{=}} sitting.i \rightarrow \textsc{Skip} \\
Sleeping(i) &\mathrel{\widehat{=}} laydown \rightarrow sleeping \rightarrow \textsc{Skip} \\
GetUp(i, j) &\mathrel{\widehat{=}} leavebed.j.i \rightarrow \textsc{Skip}
\end{aligned}
$$

Sensor Modalities

Various sensors are deployed in the environment to capture targeted activities. Sensors send their current status to the control system via a wireless network. Sensors are used to monitor

the changes in the environment. In real life, sensors are sending signals representing their status in certain interval according to their frequencies. Note that the details about signal encoding/decoding and transmission between sensors and system are related to hardware components, which are fairly reliable. Therefore we abstract out these details in our model to simplify the modeling as well as the verification.

Before modeling the sensor behaviors, we briefly introduce different sensors and their working mechanism which are used in our system.

RFID Radio-frequency identification (RFID) is a technology that uses communication via electromagnetic waves to exchange data between a terminal and an object such as a product, animal, or person for the purpose of identification and tracking. Some tags can be read from several meters away and beyond the line of sight of the reader. In our project, RFID sensors are mainly used to:

- To detect the presence of subject at specific location
- To identify and infer who is doing what activity in the environment

Reed Switch is a small electromechanical device having two ferromagnetic reeds that are hermetically sealed in a glass envelope. It is mainly used to:

- To detect the opening and closing status of the door, cabinet and drawers
- To infer the direction of entering/leaving with PIR

Motion Sensing (PIR) A Passive InfraRed sensor (PIR sensor) is an electronic device that measures infrared (IR) light radiating from objects in its field of view. PIR sensors are often used in the construction of PIR-based motion detectors (see below). Apparent motion is detected when an infrared source with one temperature, such as a human, passes in front of an infrared source with another temperature, such as a wall. In our project, PIR sensors are used to:

- To detect the presence of inhabitant at particular location

- To detect whether the inhabitant makes movements or not (coarse-grained)

Shake Sensing(Vibration)

- To detect the usage of water from inlet pipe (tap open/tap close)
- To infer the correct/incorrect activities inside bathroom with location information

Range Sensing (Infrared/Ultrasonic) To localize/detect the presence of subject/object at specific location

Pressure Sensing (FSR) A Force-sensing resistor (FSR) is a material whose resistance changes when a force or pressure is applied. In this project, we place a set of FSR under the mattress of each bed to detect various activities in bed. The detailed usage of FSR is:

- To detect the coarse-grained activities of the subject on the bed
- To activate/deactivate the RFID sensing according to the events detected from pressure sensing push

In the modeling, each sensor is modeled as a process which is triggered by residence activities and hence send signal to the control station. Process *Sensors* is an interleaving of all individual sensor modeled in the system. *DoorRFIDReader* is a RFID sensor placed on the bed room door to capture entering/leaving behavior of each residence. *BedPressureS* is a pressures sensor place on each bed to detect residence activities on the bed. Once it detects pressure sensing push on the bed, it would activate the *BedRFID* sensor to detect who is on the bed. The three sensors are modeled as follows where the complete model of all sensor behaviors are listed in Appendix C.

$$
\begin{aligned}
&\mathit{DoorRFIDReader} \mathrel{\widehat{=}} \\
&\qquad (\mathit{EnterDoorRFID} \,\Box\, \mathit{LeaveDoorRFID});\ \mathit{DoorRFIDReader} \\
&\mathit{EnterDoorRFID} \mathrel{\widehat{=}} \\
&\qquad \mathit{enterRoom}.0 \rightarrow \mathit{rfid\_room}!(0, \mathit{outside}) \rightarrow \textsc{Skip} \\
&\qquad \Box\ \mathit{enterRoom}.1 \rightarrow \mathit{rfid\_room}!(1, \mathit{outside}) \rightarrow \textsc{Skip} \\
&\qquad \textsc{Where}\ \mathit{enterRoom}.0.\textsc{Engage} = \mathit{rfid\_room}!(0, \mathit{outside}).\textsc{Engage} \\
&\qquad \wedge\ \mathit{enterRoom}.1.\textsc{Engage} = \mathit{rfid\_room}!(1, \mathit{outside}).\textsc{Engage} \\
&\mathit{LeaveDoorRFID} \mathrel{\widehat{=}} \\
&\qquad \mathit{leaveRoom}.0 \rightarrow \mathit{rfid\_room}!(0, \mathit{inRoom}) \rightarrow \textsc{Skip} \\
&\qquad \Box\ \mathit{leaveRoom}.1 \rightarrow \mathit{rfid\_room}!(1, \mathit{inRoom}) \rightarrow \textsc{Skip} \\
&\qquad \textsc{Where}\ \mathit{leaveRoom}.0.\textsc{Engage} = \mathit{rfid\_room}!(0, \mathit{inRoom}) \\
&\qquad\quad \wedge\ \mathit{leaveRoom}.1.\textsc{Engage} = \mathit{rfid\_room}!(1, \mathit{inRoom}).\textsc{Engage} \\
&\mathit{BedPressureS}(i, j) \mathrel{\widehat{=}} \\
&\qquad \mathit{sitonbed}.j.i \rightarrow \mathit{bed\_pressure\_sensor}!(j, 0) \rightarrow \mathit{BedRFID}(i, j) \\
&\qquad \Box\ \mathit{laydown}.j.i \rightarrow \mathit{bed\_pressure\_sensor}!(j, 1) \rightarrow \textsc{Skip} \\
&\qquad \Box\ \mathit{leavebed}.j.i \rightarrow \mathit{BedRFID}(-1, j) \\
&\qquad \textsc{Where}\ \mathit{sitonbed}.j.i.\textsc{Engage} = \mathit{bed\_pressure\_sensor}!(j, 0).\textsc{Engage} \\
&\qquad\quad \wedge\ \mathit{laydown}.j.i.\textsc{Engage} = \mathit{bed\_pressure\_sensor}!(j, 1).\textsc{Engage} \\
&\mathit{BedRFID}(i, j) \mathrel{\widehat{=}} \mathit{bed\_rfid}!j.i \rightarrow \textsc{Skip} \\
&\mathit{BedPressureSensor} \mathrel{\widehat{=}} \\
&\qquad (\mathit{BedPressureS}(0, 0) \,\Box\, \mathit{BedPressureS}(0, 1)) \;|||\; \\
&\qquad (\mathit{BedPressureS}(1, 0) \,\Box\, \mathit{BedPressureS}(1, 1)); \\
&\qquad \mathit{BedPressureSensor}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Sensors} \mathrel{\widehat{=}} \mathit{SRDoorRFIDReader} \;|||\; \mathit{ShowerSensor\_Tap} \;|||\; \mathit{BedPressureSensor} \\
&\qquad |||\; \mathit{DoorRFIDReader} \;|||\; \mathit{WashSensor\_Tap}
\end{aligned}
$$

Controller and Reminding Service

When a sensor detects an activity in the environment, it will send signal to the control system and then the system receives the signal and then make corresponding response to the change. The main functions of the controller are i) updating system status, ii) activate/deactivate related reminders. *LocationMonitor* receives signals from RFID sensor placed at the bed room door and then updates the system of the status of the location of each residence. *ShowerTapMonitor* receives signals from the shower tap shake sensor and then updates the *on*/*off* status of the shower tap. *BedMonitor* monitors the bed pressure sensor and bed RFID sensor and updates the bed activities as well as the occupation of each bed. The

three monitors are modeled as follows where the complete model of all monitors are listed in Appendix C.

$$
\begin{aligned}
\mathit{LocationMonitor} \quad &\widehat{=} (\mathit{rfid\_room}?(i, \mathit{status}) \rightarrow \\
&\quad [\mathit{status} == \mathit{outside}]\mathit{detectEnterRoom}.i \rightarrow \textsc{Skip} \\
&\quad \Box\, [\mathit{status} == \mathit{inRoom}]\mathit{detectLeaveRoom}.i \rightarrow \textsc{Skip}); \\
&\quad \mathit{LocationMonitor} \\
\mathit{ShowerTapMonitor} \quad &\widehat{=} (\mathit{showerroom\_tap}?1 \rightarrow \mathit{tapIsOn} \rightarrow \textsc{Skip} \\
&\quad \Box\, \mathit{showerroom\_tap}?0 \rightarrow \mathit{tapIsOff} \rightarrow \textsc{Skip}); \\
&\quad \mathit{ShowerTapMonitor} \\
\mathit{BedMonitor} \quad &\widehat{=} (\mathit{bed\_pressure\_sensor}?(j, s) \rightarrow \mathit{bed\_rfid}?(i, j) \rightarrow \\
&\quad [i! = -1 \wedge i! = j]\mathit{prompt\_sleepInWrongBed}.i \rightarrow \textsc{Skip} \\
&\quad \Box\, [i! = -1 \wedge i == j]\mathit{prompt\_goodnight}.i \rightarrow \textsc{Skip}); \\
&\quad \mathit{BedMonitor} \\
\mathit{Monitor} \quad &\widehat{=} \mathit{LocationMonitor} \;|||\; \mathit{ShowerRoomRFID} \;|||\; \mathit{ShowerTapMonitor} \\
&\quad |||\; \mathit{BedMonitor} \;|||\; \mathit{WashTapMonitor};
\end{aligned}
$$

In this project, we studied the daily behaviors of residences of the nursing home and finally choose to implement six reminders. The reminders are customized under supervising. Detailed descriptions of each reminder are as listed as follows:

Showering For Too Long Reminder Once the system detects a residence has been showering for too long (e.g., more than 1 hour), it will send a reminder to him/her/nurse to finish the shower and leave the shower room. If an elderly with mild dementia has been showering for too long, is likely that he falls down or gets stuck in the shower room; but it is not possible for the nurses to check each shower room from time to time. The showering for too long reminder is of special importance in terms of safety issue.

Sleeping in Wrong Bed Reminder Once the system detects a residence sleeps on other residence's bed, it will send a reminder to him/her to sleep on his/her own bed. It is important because most of time, when a residence is sleeping on another residence's bed, they are very likely to argue withe each other.

Flushing Toilet Reminder Once system detects a residence forgets to flush toilet after using the toilet, the system will send a reminder to him/her to flush toilet. Elderly

with mild dementia is very likely to forget flushing after toilet which results in serious hygiene issue.

Turnoff Shower Tap Reminder Once system detects a residence forgets to turnoff water tap after showering, it will send a reminder to him/her to turnoff the shower tap.

Turnoff Wash Basin Tap Reminder Once the system detects a residence leaves the wash basin tap on while finishing washing hands, it will send a reminder to him/her to turnoff the water tap.

Wake Up Reminder It is a morning wake-up alarm for each residence if they are required to wake up by some time.

Selected reminders, more specifically, *Sleep in wrong bed reminder*, *showering for too long reminder* and *flush toilet reminder* , are specified as follows, where the complete model is shown in Appendix C.

$$
\begin{aligned}
&Reminder\_SleepingInWrongBed \mathrel{\widehat{=}} \\
&\quad (prompt\_sleepInWrongBed.0 \rightarrow prompt!(wrong\_bed, 0) \rightarrow \textsc{Skip} \\
&\quad \Box\ prompt\_sleepInWrongBed.1 \rightarrow prompt!(wrong\_bed, 1) \rightarrow \textsc{Skip}); \\
&\quad Reminder\_SleepingInWrongBed \\
&\quad \textsc{Where}\ prompt!(wrong\_bed, 0).Engage - \\
&\qquad prompt\_sleepInWrongBed.0.Engage \leq 60 \wedge prompt!(wrong\_bed, 1).Engage \\
&\qquad - prompt\_sleepInWrongBed.1.Engage \leq 60 \\
&Reminder\_Flushing(duration) \mathrel{\widehat{=}} \\
&\quad (start\_toilet \rightarrow finish\_toilet \rightarrow \textsc{Skip} \overset{duration}{\triangleright} \\
&\quad flush \rightarrow prompt_f lushtoilet \rightarrow \textsc{Skip}); \\
&\quad Reminder\_Flushing(duration) \\
&Reminder\_ShowerLong(duration) \mathrel{\widehat{=}} \\
&\quad (detectEnterShowerRoom.0 \rightarrow \textsc{Skip} \overset{duration}{\triangleright} \\
&\qquad detectLeaveShowerRoom.0 \rightarrow prompt!showertoolong \rightarrow \textsc{Skip} \\
&\quad \Box\ detectEnterShowerRoom.1 \rightarrow \textsc{Skip} \overset{duration}{\triangleright} \\
&\qquad detectLeaveShowerRoom.1 \rightarrow prompt!showertoolong \rightarrow \textsc{Skip}); \\
&\quad Reminder\_ShowerLong(duration)
\end{aligned}
$$

Prompting is the implementation of the prompting service which receives signals from *prompt* channel and then send alarms according to the type of reminder received from the channel. It is specified as:

$Prompting \mathrel{\widehat{=}} prompt?status \rightarrow prompting \rightarrow Prompting;$

7.2.2 Verification

With the Timed Planning model presented above, this section is devoted to the verification of various properties of the system. We categorize the properties according to the following three types.

Deadlock Freeness

Deadlock is a common undesired state for safety-critical systems, especially in our case, a smart nursing home for mild dementia patients. The existence of a deadlock state indicates there might be potential error for the system which can lead to sensor erroneous or reminder erroneous. An assertion for verifying deadlock freeness is defined in a CLP relation $deadlock(P, Tr)$, where P is the reference of the smart nursing home model and Tr is a deadlock witness trace. By executing the goal:

$? - deadlock(smartnursinghome, Tr).$

Reminder Correctness

The most important (safety) properties that we want to preserve are that whenever the critical situation occurs, the reminders will work as designed. For example, if the shower tap is left on and nobody is inside the shower room, will the *Turnoff Shower Tap Reminder* be sent out on time. It's important to check the prompting of reminders based on time. To check the correctness of each reminder, we list the following properties as follows.

- **FTRP**: After a residence has finished toilet for sometime and no flushing action has been taken, whether the *flushing toilet reminder* would prompt within 1 minute (60 seconds) or not?

Property	Result	Execution Time
Deadlock-freeP	true	421
FTRP	true	12
STLRP	true	5
SWBRP	true	20
TSTRP	true	13
TWTRP	true	25
FalseAlarmP1	true	5
FalseAlarmP2	true	7

Table 7.2: Results of Experiment

- **STLRP**: If a residence is showering for more than the maximum time, will *showering for too long reminder* prompt in 1 minute?
- **SWBRP**: If a residence sleeps in a bed which not his, whether the *sleeping in wrong bed reminder* would prompt in 30 seconds?
- **TSTRP**: Once a residence exits the shower room and leaves shower tap on, will *turnoff shower tap reminder* would prompt in 1 minute?
- **TWTRP**: Once a residence exits the washing basin area and leaves the washing tap on, whether the *turnoff wash basin tap reminder* would be prompted or not?

Experiments

In this section, we discuss how each component is modeled as Timed Planning processes and how the properties are defined as CLP goals. We test seven properties defined previously among the Smart Nursing Home system on Windows XP with a 2.0 GHz Intel CPU and 2 GB memory. The result is shown in Table 7.2.

7.2.3 Modeling with PAT

To compare with PAT model checker, we model the smart nursing home system in PAT's syntax and verify the same properties.

Since the modeling language of PAT is also an extension of Timed CSP, the PAT module of smart nursing home system is very similar to the Timed Planning model described in the previous section, hence ignored here. The complete PAT model of the smarting home system is shown in Appendix D. However, we remark that there are two differences in the modeling. Firstly, PAT supports global variables but not Timed Planning. Therefore, Timed Planning model of smart nursing home system adopts a different approach in modeling the status changes of the sensors. Secondly, Timed Planning supports WHERE clause but not in PAT. Therefore, PAT model uses the timeout or ranged wait or within operators to achieve the same effects.

For the properties presented in Section 7.2.2, PAT can verify only partial of them due to the different approach in verifying the system. On the other hand, some LTL properties that can be verified by PAT, but hardly encoded in the CLP syntax. We list some LTL properties below that can be checked by PAT only.

- **SWBRP**: [](tapon_nobodyin -> <>prompt_TurnOffTap)
- **TSTRP**: [](laydown.0.1 -> <> prompt_sleepInWrongBed.0)
- **TWTRP**: [](enterShowerRoom.0 || enterShowerRoom.1 -> <> showertapon)

In summary, PAT and Timed Planning provides different ways to modeling and verifying the real-time systems. Their modeling language is very similar and can be used to model hierarchical real-time systems. PAT supports the global variables, which can make the modeling easier sometimes. For the verification support, PAT can support safety properties, LTL properties and refinement checking. Timed Planning can verify properties related to time.

7.3 Summary

Pervasive computing techniques have been proposed to assist elders with mild dementia to improve their level of independence and quality of life through cognitive reinforcement. In this chapter, we demonstrate how to support formal analysis of pervasive computing systems by using Timed Planning to model and verify critical properties.

To demonstrate the feasibility of our approach, we use model and verify systems in two different settings. Firstly,a context-aware reminding framework for elders living at home alone is built and modeled using Timed Planning specification. Secondly, a reminding framework for elders living at nursing home is modeled using Timed Planning. The first case study focuses on the various reminders and conflicts between reminders. The second case study focuses on the various sensors and reasoning about the rules. For the second case study, we compare our approach with the PAT modeling techniques. The results shows that PAT has better performance but cannot verify the properties related to timing, while our approach has the advantage in expressing complication property. The future work for us is to improve the performance of the approach by adding more reduction and optimization techniques.

Chapter 8

Conclusion

This chapter concludes the thesis. Section 8.1 summarizes the contribution of this thesis and Section 8.2 discusses some on-going and future directions.

8.1 Summary of the Thesis

In this thesis, we focused on the modeling and verification of real-time systems by using timed process algebra. In particular, we have tried to address four issues related to real-time systems modeling and verification.

1. Proposing a formal language for modeling real-time systems with hierarchical strutures,
2. Exploring efficient verification techniques to perform formal analysis of the proposed models,
3. Implementing a toolkit to support effective real-time verification, and
4. Applying the proposed techniques in different domains.

Timed CSP is a timed extension of CSP, which has been proposed for decades. Such specification languages are elegant and intuitive as well as precise, which have been widely accepted and applied to a wide arrange of systems. The first piece of work we have done is building a reasoning mechanism for a timed process algebra, Timed CSP. The chosen underlying reasoning support is Constraint Logic Programming (CLP). We have shown that event-based formalism like Timed CSP can be encoded in CLP by following its semantics. Our approach therefore broadened real-time systems which can be specified and verified by CLP. This setups a solid foundation for extending Timed CSP with more expressive operators and developing a reasoning engine for this extended Timed CSP specification. In details, our approach started with a formal translation of Timed CSP syntax to CLP relations, where a library of all Timed CSP operators is built. Next, a collection of supplementary rules for translating the operational semantics of each Timed CSP operator into CLP relations were defined. We carried out verification of Timed CSP models with a high level of automation. We investigated a wide range of properties that may be proved based on constraint solving, namely CLP($\mathcal{R}$), for instance we showed that using a unique interpretation, safety properties and liveness properties can be proved effectively as well as properties such as lower or upper bound of variables and timewise refinement. A prototype tool HORAE, which is an interactive software that provides composing and reasoning of Timed CSP models.

The second piece of work in the thesis is to propose a new formalism for real-time systems by extending Timed CSP. This extension names Timed Planning. Due to the fact that the semantics of Timed CSP lacks of support for stating system requirements which constraints all behavioral traces of given processes, for example, *deadline* and execution time of a process, time-related constraints among events which are common requirements for many real time systems and etc. Timed Planning extends Timed CSP with the capability of stating complicated timing behaviors for processes and events to model and verify complex compositional real-time systems. A Timed Planning model is made up of a compositional timed process and a set of constraints over processes, events and the data variables which are the requirements that the process should satisfy. In this approach each process is associated with a set of localized timing/untiming requirements with keyword WHERE, which can be

specified in a compositional way. The Timed Planning provided a framework to specify a number of compositional timed/untimed behaviors to capture various requirements over the system. It has more expressive power than Timed Automata, for example it has the capability of keeping timed event set during execution and applying some operations on the set. In this thesis, we formally defined complete syntax and operational semantics of Timed Planning. Due to the expressive power, CLP can express the syntax and semantics of Timed Planning completely. A mechanized proving system, based on CLP($\mathcal{R}$), has been developed by extending the reasoning mechanism for Timed CSP. For verifying systems built using Timed Planning specification, a set of rules for feasibility testing are defined to find the conflict constraints among processes. Besides, various safety and liveness properties can also be verified over systems modeled in Timed Planning. We believe that Timed Planning is not just a syntax sugar over Timed CSP, but it has more expressive power than Timed CSP.

An important goal of formal specification languages is to solve problems in various domains. In the thesis, we applied Timed Planning in three different domains, namely, scheduling, security protocols and pervasive computing. Firstly, we applied Timed Planning and its reasoning engine to solve classical job-shop scheduling problems, where finding an optimal schedule corresponds to finding a shortest execution (in terms of elapsed time) in the Timed Planning. In our approach, the job-shop scheduling problem was naturally modeled as Timed Planning processes. We also worked with the extended job-shop scheduling problems, where all jobs have composition operational behaviors. Besides, jobs with deadline and relative timing constrains are also able to be captured in our approach. We believe that the insight gained from this point of view will contribute both to scheduling and to the study of timed processes. We have demonstrated that the performance of the Timed Planning approach of solving job-shop scheduling problem can be highly improved by applying a set of optimizations. There are still many potential improvements to be explored to reduce the execution time, such as new partial-order methods and heuristics, etc.

Another piece of work is we proposed a new method of modeling and analyzing timed security protocols which consists of various timing aspects such as timestamps, delays, timeouts and a set of timing constraints. To fulfill the aim, we substantially extended Timed Planning with capabilities to stating complicated and critical timing requirements of timed security protocols, in a compositional way. Based on our previous work on building a reasoning tool for Timed CSP, a prototype mechanized proving system, based on CLP($\mathcal{R}$) to verify various properties over systems modeled in this extended specification has been built. We model principals as processes, as well as the cryptograph device, including timestamps, timeout, retransmissions and delays. The timed non-injective agreement authentication property can be verified using our underlying reasoning engine, which can be easily extended to verify other authentication properties. We propose a novel approach to find timing attacks using timing information of protocol sessions. We also modeled timing requirements of the protocols in our WHERE predicates and verify other timing properties of the protocols.

8.2 Future Works

Based on this thesis, there are a number of directions for future work that may be beneficial to the verification system of timed process algebra languages and the applications. In this section, some of these possible research directions are briefly discussed.

8.2.1 Reduction and Optimization Techniques

First, we plan to investigate methods to combine well-known state space reduction techniques. We know that systems that accept an infinite number of threads or unbound data structures make model checking impossible. Symmetric properties among threads can reduce infinite number of threads to a small number. Data abstraction for infinite domain data variables can also be incorporated into the model checking to handle unbounded data size. Furthermore, combining the effective search heuristics in the CLP(R) with the verifi-

cation algorithm is always attractive for us. These solutions are valuable for our verification algorithms.

8.2.2 New Application Domains

As a future work, we will apply Timed Planning specifications to other domains, such as timed scheduling problems, complex real-time systems and etc. For analyzing timed security protocols, we will expand the verification of security properties, such as secrecy, integrity, fairness and the timing properties such as the time range for an easy attack. For our underlying reasoning engine, there are many potential improvements to be explored to reduce the execution time, such as symmetry reduction and heuristics.

8.2.3 Tool Development

To perform efficient verification, tool implementation is the key. Based on the current prototype, we shall improve it in the following directions. Firstly, a user friendly GUI can help users to input the model easily. Secondly, integrate the new version of CLP(R) into the system. The current CLP(R) we are using is based on the IBM version released at 1992. The new version of CLP(R) is under active development with a lot of bug fixing and performance improvement. To adopt the new CLP(R) and investigate the new reduction technique can be very effective for the verification of real-time systems. Lastly, we want to formally define the property patterns, which could help users easily come up with the predicates for properties.

Bibliography

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

[2] Y. Abdeddaïm and O. Maler. Job-shop scheduling using timed automata. In *CAV '01*, pages 478–492, 2001.

[3] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 113–126, London, UK, 2002. Springer-Verlag.

[4] R. Abhik and I. Ramakrishnan. Automated Inductive Verification of Parameterized Protocols. In *International Conf. on Computer Aided Verification (CAV)*. Springer, 2001.

[5] R. Alur and D. L. Dill. A Theory of Timed Automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] R. Anderson and R. Needham. Programming satan's computer. In *in Computer Science Today*, pages 426–440. Springer-Verlag, 1995.

[7] G. Bella and L. C. Paulson. Kerberos version iv: Inductive analysis of the secrecy goals. In *ESORICS 98, LNCS 1485*. Springer, 1998.

[8] M. Boreale and M. G. Buscemi. Experimenting with sta, a tool for automatic analysis of security protocols. In *SAC '02: Proceedings of the 2002 ACM symposium on Applied computing*, pages 281–285, New York, NY, USA, 2002. ACM.

[9] L. Bozga, Y. Lakhnech, and M. Périn. Pattern-based abstraction for verifying secrecy in protocols. *Int. J. Softw. Tools Technol. Transf.*, 8(1):57–76, 2006.

[10] M. Bozga, C. Daws, O. Maler, A. Olivero, S. Tripakis, and S. Yovine. Kronos: A Model-Checking Tool for Real-Time Systems. In *CAV 1998*, volume 1427 of *LNCS*, pages 546–550. Springer, 1998.

[11] P. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, 1999.

[12] A. B. Bruno Bouchard and S. Giroux. A keyhole plan recognition model for alzheimerąŕs patients: First results. *Journal of Applied Artificial Intelligence (AAI)*, 22:S10–5, 2007.

[13] M. Burrows, M. Abadi, and R. Needham. A logic of authentication. *ACM Transactions on Computer Systems*, 8:18–36, 1990.

[14] D. Cook and S. Das. How smart are our environments? an updated look at the state of the art. *Journal of Pervasive and Mobile Computing*, 2007).

[15] R. Corin and S. Etalle. An improved constraint-based system for the verification of security protocols. In *SAS 2002*, pages 326–341, 2002.

[16] R. Corin, S. Etalle, P. H. Hartel, and A. Mader. Timed analysis of security protocols. *J. Comput. Secur.*, 15(6):619–645, 2007.

[17] J. Davies. *Specification and Proof in Real-Time CSP*. Cambridge University Press, 1993.

[18] J. Davies and S. Schneider. A Brief History of Timed CSP. *Theor. Comput. Sci.*, 138, 1995.

[19] G. Delzanno and S. Etalle. Proof theory, transformations, and logic programming for debugging security protocols. In *LOPSTR 2001*, pages 76–90, 2001.

[20] G. Delzanno and P. Ganty. Automatic verification of time sensitive cryptographic protocols. In *TACAS 2004*.

[21] D. L. Dill. Timing Assumptions and Verification of Finite-State Concurrent Systems. In *Automatic Verification Methods for Finite State Systems*, pages 197–212, 1989.

[22] D. Dolev and A. C. Yao. On the security of public key protocols. In *SFCS '81*, pages 350–357, Washington, DC, USA, 1981. IEEE Computer Society.

[23] J. S. Dong, Y. Feng, J. Sun, and J. Sun. Sensor based design techniques for smart system space. In *ICMOCCA '06: Proceedings of the First International Conference on Mobile Computing, Communications and Applications*, August 2006.

[24] J. S. Dong, P. Hao, S. Qin, and X. Zhang. The semantics and tool support of ozta. In *Proceedings of the 7th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2005)*, pages 66–80, 2005.

[25] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and Y. Wang. Timed Patterns: TCOZ to Timed Automata. In J. Davies, W. Schulte, and M. Barnett, editors, *Proceedings of the 6th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2004)*, volume 3308 of *Lecture Notes in Computer Science*, pages 483–498. Springer, 2004.

[26] J. S. Dong, P. Hao, S. C. Qin, J. Sun, and W. Yi. Timed Automata Patterns. *IEEE Transactions on Software Engineering*, 34(6):844–859, 2008.

[27] J. S. Dong, P. Hao, J. Sun, and X. Zhang. Reasoning about Timed CSP Models . In *FM 2006*. http://fm06.mcmaster.ca/research_exhibition.htm.

[28] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *Proceedings of the 8th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *LNCS*, pages 342–359. Springer, 2006.

[29] J. S. Dong, P. Hao, J. Sun, and X. Zhang. A Reasoning Method for Timed CSP Based on Constraint Solving. In *ICFEM 2006*, pages 342–359, 2006.

[30] J. S. Dong, P. Hao, X. Zhang, and S. Qin. Highspec: a tool for building and checking ozta models. In *Proceedings of the 28th International Conference on Software Engineering (ICSE 2006)*, pages 775–778, 2006.

[31] J. S. Dong, Y. Liu, J. Sun, and X. Zhang. Verification of Computation Orchestration Via Timed Automata. In *Proceedings of the 8th International Conference on Formal Engineering Methods (ICFEM 2006)*, volume 4260 of *LNCS*, pages 226–245. Springer, 2006.

[32] J. S. Dong, B. P. Mahony, and N. Fulton. Modeling Aircraft Mission Computer Task Rates. In *FM 1999*, page 1855, 1999.

[33] J. S. Dong, J. Sun, J. Sun, K. Taguchi, and X. Zhang. Specifying and verifying sensor networks: an experiment of formal methods. In *Proceedings of the 10th International Conference on Formal Engineering Methods (ICFEM 2008)*, volume 5256 of *Lecture Notes in Computer Science*, pages 318–337. Springer, Oct 2008.

[34] J. S. Dong, J. Sun, and X. Zhang. Reasoning of Timed CSP and Extensions. In *Technical report*. http://www.comp.nus.edu.sg/∼zhangxi5/tp.pdf.

[35] K. Du, D. Q. Zhang, M. Musa, M. Mokhtari, and X. Zhou. Handling activity conflicts in reminding system for elders with dementia. In *FGCN 2008 :Future Generation Communication and Networking*, volume 2, pages 416 – 421, 2008.

[36] K. Du, D. Q. Zhang, X. Zhou, M. Musa, M. Mokhtari, and W. Qin. Hycare: A hybrid context-aware reminding framework for elders with mild dementia. In *ICOST 2008 :6th International Conference on Smart Homes and Health Telematics*, pages 9 – 17, 2008.

[37] X. Du, C. R. Ramakrishnan, and S. A. Smolka. Tabled resolution + constraints: A recipe for model checking real-time systems. In *In IEEE Real Time Systems Symposium (RTSS*, pages 175–184. Society Press, 1999.

[38] C. N. et al. Home based assistive technologies for people with mild dementia. In *ICOST 2007 : 5th International Conference on Smart Homes and Health Telematics*, volume 4541/2007, pages 63–69, Nara, Japan, 2007. Springer.

[39] N. Evans and S. Schneider. Analysing time dependent security properties in csp using pvs. In *ESORICS '00*, pages 222–237, London, UK, 2000. Springer-Verlag.

[40] D. A. Fisher. The interaction between the preliminary designs and the technical requirements for the dod common high order language. In *ICSE 1978*, page 82ÍC83, 1978.

[41] M. L. Fisher. Optimal solution of scheduling problems using lagrange multipliers: Part i. *OPERATIONS RESEARCH*, 21:1114–1127, 1973.

[42] Formal Systems (Europe) Ltd. Failure Divergence Refinement: FDR2 User Manual. 1997.

[43] L. Fratiglioni, L. J. Launer, K. Andersen, M. M. Breteler, J. R. Copeland, J. F. Dartigues, A. Lobo, J. Martinez-Lage, H. Soininen, and A. Hofman. Prevalence of dementia and major subtypes in europe: A collaborative study of population-based cohorts. neurologic diseases in the elderly research group. *Neurology*, 54(11 Suppl 5):S4–9, 2000.

[44] L. Fratiglioni, L. J. Launer, K. Andersen, and etc. Incidence of dementia and major subtypes in europe: A collaborative study of population-based cohorts,neurol dis elderly res group. neurology. *Neurology*, 54:S10–5, 2000.

[45] C. R. R. G. Pemmasani and I. V. Ramakrishnan. Efficient real-time model checking using tabled logic programming and constraints. In *ICLP 2002: 18th International Conference on Logic Programming*, pages 100–114. Springer, 2002.

[46] G.Lowe. A hierarchy of authentication specifications. In *CSFW'97*, pages 31–44, 1997.

[47] T. Göthel and S. Glesner. Machine Checkable Timed CSP. In *NFM 2009*. NASA Conference Publication, 2009.

[48] G. Gupta and E. Pontelli. A Constraint-based Approach for Specification and Verification of Real-time Systems. In *RTSS '97*, page 230, 1997.

[49] D. Harel and E. Gery. Executable Object Modeling with Statecharts. *IEEE Computer*, 30(7):31–42, 1997.

[50] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. *Information and Computation*, 111(2):193–244, 1994.

[51] C. A. R. Hoare. *Communicating Sequential Processes.* International Series in Computer Science. Prentice-Hall, 1985.

[52] C. A. R. Hoare. *Communicating Sequential Processes.* International Series on Computer Science. Prentice-Hall, 1985.

[53] J. Jaffar and J. Lassez. Constraint Logic Programming. In *POPL*, pages 111–119, 1987.

[54] J. Jaffar and M. J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming,1994.*

[55] J. Jaffar, S. Michaylov, P. J. Stuckey, and R. H. C. Yap. The CLP(R) Language and System. *ACM Trans. Program. Lang. Syst.*, 14(3):339–395, 1992.

[56] J. Jaffar, A. E. Santosa, and R. Voicu. A CLP Proof Method for Timed Automata. In *Real-Time Systems Symposium*, pages 175–186, 2004.

[57] J. Jaffar, A. E. Santosa, and R. Voicu. Modeling Systems in CLP. In *ICLP'05*, 2005.

[58] A. Jain and S. Meeran. Deterministic job-shop scheduling: Past, present, and future. In *European Journal of Operational Research 113 (2)*, pages 390–434, 1999.

[59] J. J.R. Absher. Dementia diagnosis and therapy in the elderly. In *Compr Ther 1992*, pages 28 – 33, 1992.

[60] L. M. Lai and P. Watson. A Case Study in Timed CSP: The Railroad Crossing Problem. In *HART 1997*, pages 69–74, 1997.

[61] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, 3:125–143, 1977.

[62] K. G. Larsen, P. Pettersson, and Y. Wang. Uppaal in a Nutshell. *International Journal on Software Tools for Technology Transfer*, 1(1-2):134–152, 1997.

[63] M. Lindahl, P. Pettersson, and Y. Wang. Formal Design and Analysis of a Gearbox Controller. *STTT 2001*, 3(3):353–368, 2001.

[64] Y. Liu. *Model Checking Concurrent and Real-time Systems: the PAT Approach.* PhD thesis, National University of Singapore, 2009.

[65] Y. Liu, W. Chen, Y. A. Liu, and J. Sun. Model Checking Linearizability via Refinement. In *FM 2009*, pages 321–337, 2009.

[66] Y. Liu, J. Sun, and J. S. Dong. An analyzer for extended compositional process algebras. In *ICSE Companion*, pages 919–920. ACM, 2008.

[67] Y. Liu, J. Sun, and J. S. Dong. Scalable Multi-Core Model Checking Fairness Enhanced Systems. In *ICFEM 2009*, pages 426–445, 2009.

[68] Y. Liu, J. Sun, and J. S. Dong. Developing model checkers using pat. In A. Bouajjani and W.-N. Chin, editors, *Automated Technology for Verification and Analysis - 8th International Symposium, ATVA 2010, Singapore, September 21-24, 2010. Proceedings*, volume 6252 of *Lecture Notes in Computer Science*, pages 371–377. Springer, 2010.

[69] Y. Liu, J. Sun, and J. S. Dong. Pat 3: An extensible architecture for building multi-domain model checkers. In *ISSRE*, pages 190–199, 2011.

[70] G. Lowe. Breaking and fixing the needham-schroeder public-key protocol using fdr. In *TACAs '96: Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 147–166, London, UK, 1996. Springer-Verlag.

[71] N. A. Lynch and F. W. Vaandrager. Action Transducers and Timed Automata. *Formal Aspects of Computing*, 8(5):499–538, 1996.

[72] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: An Introduction to TCOZ. In *Proceedings of the 20th International Conference on Software Engineering (ICSE 1998)*, pages 95–104, Kyoto, Japan, 1998.

[73] B. P. Mahony and J. S. Dong. Network Topology and a Case Study in TCOZ. In *Proceedings of the 11th International Conference of Z Users (ZUM 1998)*, volume 1493 of *LNCS*, pages 308–327, 1998.

[74] B. P. Mahony and J. S. Dong. Timed Communicating Object Z. *IEEE Trans. Software Eng.*, 26(2):150–177, 2000.

[75] D. May. OCCAM. *SIGPLAN Notices*, 18(4):69ÍC79, 1983.

[76] F. J. M. Merliand, D. Craig, F. Molaert, M. Mulvenna, C. Nugent, T. Scully, J. Bengtsson, and R. M. Droses. Helping people with mild dementia navigate their day. In *ICMCC 2007 :International Council on Medical and Care Compunetics*, Amsterdam , Netherland, 2007.

[77] P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, Department of Information and Computer Science, University of California, 1974.

[78] T. K. Nguyen, J. Sun, Y. Liu, and J. S. Dong. A model checking framework for hierarchical systems. In *ASE*, pages 633–636, 2011.

[79] X. Nicollin and J. Sifakis. The Algebra of Timed Processes, ATP: Theory and Application. *Information and Computation*, 114(1):131–178, 1994.

[80] C. L. Pape and P. Baptiste. An experimental comparison of constraint-based algorithms for the preemptive job-shop scheduling problem. In *CP97 Workshop on Industrial COnstraint-Directed Scheduling.*

[81] N. M. R. Khalaf and S. Weerawarana. Service-Oriented Composition in BPEL4WS. In *WWW 2003*, 2003.

[82] G. M. Reed and A. W. Roscoe. A Timed Model for Communicating Sequential Processes. In L. Kott, editor, *ICALP*, volume 226 of *Lecture Notes in Computer Science*, pages 314–323. Springer, 1986.

[83] J. H. Reppy. CML: A Higher-Order Concurrent Language. In *PLDI 1991*, page 293ÍC305, 1991.

[84] A. W. Roscoe. Model-checking CSP. *A classical mind: essays in honour of C. A. R. Hoare*, pages 353–378, 1994.

[85] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall, 1997.

[86] A. W. Roscoe, P. H. B. Gardiner, M. Goldsmith, J. R. Hulance, D. M. Jackson, and J. B. Scattergood. Hierarchical Compression for Model-Checking CSP or How to Check 10^{20} Dining Philosophers for Deadlock. In *Proceedings of the 1st International Conference of Tools and Algorithms for Construction and Analysis of Systems (TACAS 1995)*, pages 133–152, 1995.

[87] P. Ryan, S. Schneider, M. Goldsmith, G. Lowe, and B. Roscoe. Modelling and Analysis of Security Protocols, 2001.

[88] S. Schneider. *Concurrent and Real-time System: The CSP Approach*. JOHN WILEY & SONS, LTD, 2000.

[89] S. Schneider. *Concurrent and Real-time Systems: the CSP Approach*. John Wiley and Sons, 2000.

[90] S. A. Schneider. An Operational Semantics for Timed CSP. In *Proceedings Chalmers Workshop on Concurrency, 1991*, pages 193 – 213.

[91] A. P. Sistla and E. Clarke. The Complexity of Propositional Temporal Logics. *The Journal of ACM*, 32:733–749, 1986.

[92] G. Smith and J. Derrick. Specification, Refinement and Verification of Concurrent Systems-An Integration of Object-Z and CSP. *Formal Methods in System Design*, 18(3):249–284, 2001.

[93] J. Sun, Y. Liu, and B. Cheng. Model checking a model checker: A code contract combined approach. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 518–533. Springer, 2010.

[94] J. Sun, Y. Liu, J. S. Dong, and C. Q. Chen. Integrating Specification and Programs for System Modeling and Verification. In *TASE 2009*, pages 127–135, 2009.

[95] J. Sun, Y. Liu, J. S. Dong, and J. Pang. Pat: Towards flexible verification under fairness. volume 5643 of *Lecture Notes in Computer Science*, pages 709–714. Springer, 2009.

[96] J. Sun, Y. Liu, J. S. Dong, and J. Pang. PAT: Towards Flexible Verification under Fairness. In *CAV*, pages 702–708, 2009.

[97] J. Sun, Y. Liu, J. S. Dong, G. Pu, and T. H. Tan. Model-based methods for linking web service choreography and orchestration. In *APSEC 2010*, pages 166 – 175, 2010.

[98] J. Sun, Y. Liu, J. S. Dong, and G. G. Pu. Model-based Methods for Linking Web Service Choreography and Orchestration. Submitted for review.

[99] J. Sun, Y. Liu, J. S. Dong, and X. Zhang. Verifying stateful timed csp using implicit clocks and zone abstraction. In K. Breitman and A. Cavalcanti, editors, *Proceedings of the 11th IEEEInternational Conference on Formal Engineering Methods (ICFEM 2009)*, volume 5885 of *Lecture Notes in Computer Science*, pages 581–600. Springer, 2009.

[100] J. Sun, Y. Liu, A. Roychoudhury, S. Liu, and J. S. Dong. Fair model checking with process counter abstraction. In A. Cavalcanti and D. Dams, editors, *Proceedings of the Second World Congress on Formal Methods (FM'09)*, volume 5850 of *Lecture Notes in Computer Science*, pages 123–139. Springer, 2009.

[101] J. Sun, Y. Liu, S. Song, J. S. Dong, and X. Li. Prts: An approach for model checking probabilistic real-time hierarchical systems. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 147–162. Springer Berlin / Heidelberg, 2011.

[102] J. Sun, S. Song, and Y. Liu. Model checking hierarchical probabilistic systems. In J. S. Dong and H. Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 388–403. Springer, 2010.

[103] T. H. Tan, Y. Liu, J. Sun, and J. S. Dong. Verification of orchestration systems using compositional partial order reduction. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 98–114. Springer Berlin / Heidelberg, 2011.

[104] S. Vurgun, M. Philipose, and M. Pavel. A statistical reasoning system for medication prompting. In J. Krumm, G. D. Abowd, A. Seneviratne, and T. Strang, editors, *Ubicomp*, volume 4717 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2007.

[105] D. S. Warren. Programming with Tabling in XSB. In *PROCOMET '98: Proceedings of the IFIP TC2/WG2.2,2.3 International Conference on Programming Concepts and Methods*, pages 5–6, London, UK, 1998.

[106] M. Winikoff, P. Dart, and E. Kazmierczak. Animating z using logic programming techniques.

[107] J. Woodcock and J. Davies. *Using Z: Specification, Refinement, and Proof*. Prentice-Hall International, 1996.

[108] W. Yi. CCS + Time = An Interleaving Model for Real Time Systems. In *Proceedings of the 18th Colloquium on Automata, Languages and Programming (ICALP 1991)*, volume 510 of *LNCS*, pages 217–228. Springer, 1991.

[109] D. Q. Zhang, M. Hariz, and M. Mokhtari. Assisting elders with mild dementia staying at home. *Pervasive Computing and Communications, IEEE International Conference on*, 0:692–697, 2008.

[110] S. J. Zhang, J. Sun, J. Pang, Y. Liu, and J. S. Dong. On combining state space reductions with global fairness assumptions. In M. Butler and W. Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lecture Notes in Computer Science*, pages 432–447. Springer Berlin / Heidelberg, 2011.

[111] X. Zhang. Job-shop scheduling problems using timed planning. In *Workshop on International Conference on Secure Software Integration and Reliability Improvement (SSIRI 2010)*, pages 66–80, 2010.

[112] X. Zhang, Y. Liu, and M. Auguston. Modeling and analyzing timed security protocols using extended timed csp. In *SSIRI 2010: International Conference on Secure Software Integration and Reliability Improvement*, pages 217–226, 2010.

[113] M. Zheng, J. Sun, Y. Liu, J. S. Dong, and Y. Gu. Towards a model checker for nesc and wireless sensor networks. In S. Qin and Z. Qiu, editors, *Formal Methods and Software Engineering*, volume 6991 of *Lecture Notes in Computer Science*, pages 372–387. Springer Berlin / Heidelberg, 2011.

Appendix A

Healthiness Conditions for Timed Planning

Implicit predicates for most of the process operators are defined as follows, where P denotes process, e denotes event, X and Y are set of events.

Event Prefix: $a^n \rightarrow P \ : \forall\, a \rightarrow P \bullet a^n.\textsc{Engage} \leqslant P.\textsc{Start}$

Sequence: $P1;\ P2 \ : \forall\, P1;\ P2 \bullet P1.\textsc{End} \leqslant P2.\textsc{Start}$

Choice: $P1 \Box P2 \ : \forall\, P1 \ ||| \ P2 \bullet P1.\textsc{Start} = P2.\textsc{Start} \wedge P1.\textsc{End} = P2.\textsc{End}$

Timeout: $P1 \triangleright \{d\}\ P2 \ : \forall\, P1 \triangleright \{d\}\ P2 \bullet \mathit{init}\{P1\}.\textsc{Engage} <= d \vee P2.\textsc{Start} = d$

Interleaving: $P1 \ ||| \ P2 \ : \forall\, P_1 \ ||| \ P_2 \bullet P_1.\textsc{Start} = P_2.\textsc{Start} \wedge P_1.\textsc{End} = P_2.\textsc{End}$

Interrupt: $P1 \bigtriangledown P2 \ : \forall\, P_1 \bigtriangledown P_2 \bullet P_1.\textsc{Start} \leqslant P_2.\textsc{Start}$

Timed Interrupt: $P1 \bigtriangledown \{d\} P2 \ : \forall\, P_1 \bigtriangledown \{d\}\ P_2 \bullet P_1.\textsc{Start} + d \leqslant P_2.\textsc{Start}$

Parallel: $P1\ {}_X||_Y\ P2 \ : \forall\, P_1\ {}_X||_Y\ P_2, \forall\, a \in X \cap Y \bullet P_1.\textsc{Start} = P_2.\textsc{Start} \wedge$
$P_1.\textsc{End} = P_2.\textsc{End} \wedge P_1.a.\textsc{Engage} = P_2.a.\textsc{Engage}$

Appendix B

Real-time Multi-lift System

$noOfFloor : \mathbb{N};\ noOfLifts : \mathbb{N}$

$$
\begin{array}{lll}
GroundFloor & \widehat{=} & request?(1, up) \rightarrow upbottomon \rightarrow enter!(1, up) \rightarrow GroundFloor \\
 & & \Box service?(1, up) \rightarrow upbottomoff \rightarrow GroundFloor \\
TopFloor & \widehat{=} & request?(noOfFloor, down) \rightarrow downbottomon \rightarrow \\
 & & \quad enter!(noOfFloor, down) \rightarrow TopFloor \\
 & & \Box service?(noOfFloor, down) \rightarrow downbottomoff \rightarrow TopFloor \\
MidFloor(level) & \widehat{=} & request?(level, up) \rightarrow upbottomon \rightarrow \\
 & & \quad enter!(level, up) \rightarrow MidFloor(level) \\
 & & \Box request?(level, down) \rightarrow downbottomon \rightarrow \\
 & & \quad enter!(level, down) \rightarrow Floor(level) \\
 & & \Box service?(level, up) \rightarrow upbottomoff \rightarrow MidFloor(level) \\
 & & \Box service?(level, down) \rightarrow downbottomoff \rightarrow MidFloor(level) \\
Floors & \widehat{=} & (|||_{1<i<noOfFloor}\ MidFloor(i)) \ ||| \ GroundFloor \ ||| \ TopFloor
\end{array}
$$

$$
\begin{array}{lll}
OpenDoor(i) & \widehat{=} & servo!(i, toOpen) \rightarrow sensor?(i, opened) \rightarrow \textsc{Skip} \\
CloseDoor(i) & \widehat{=} & servo!(i, toClose) \rightarrow sensor?(i, closed) \rightarrow \textsc{Skip} \\
CycleDoor(i) & \widehat{=} & OpenDoor(i);\ confirm \rightarrow (\mu \bullet CD\ \textsc{Wait}[t];\ CloseDoor \\
 & & \quad \triangledown\ sensor?(i, interrupt) \rightarrow OpenDoor;\ CD) \\
Door(i) & \widehat{=} & open?i \rightarrow CycleDoor(i);\ close!i \rightarrow Door(i) \\
 & & \textsc{Where}\ close.\textsc{Engage}_i - open.\textsc{Engage}_i \leqslant t_d \\
Servomech & \widehat{=} & servo?(i, toOpen) \rightarrow Servomech \\
 & & \Box servo?(i, toClose) \rightarrow Servomech \\
 & & \Box sensor!(i, opened) \rightarrow Servomech \\
 & & \Box sensor!(i, closed) \rightarrow Servomech \\
 & & \Box sensor!(i, request) \rightarrow Servomech
\end{array}
$$

$$
\begin{aligned}
&\mathit{Shalf}(i) \mathrel{\widehat{=}} \mathit{move}?n \rightarrow \mathit{arrive} \rightarrow \mathit{Shalf}(i) \\
&\qquad \textsc{Where}\ n > 0 \wedge |\ n \mid *t - \mathit{delta} \leqslant \mathit{arrive}.\textsc{Engage} - \mathit{move}.\textsc{Engage} \\
&\qquad\qquad \wedge\ \mathit{arrive}.\textsc{Engage} - \mathit{move}.\textsc{Engage} \leqslant |\ n \mid *t + \mathit{delta}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Move1}(i, \mathit{dest}, \mathit{fl}) && \mathrel{\widehat{=}} \mathit{move}!(\mathit{dest} - \mathit{fl}) \rightarrow \mathit{arrive}?\mathit{dest} \rightarrow \\
&&& \quad \mathit{open}!i \rightarrow \mathit{confirm} \rightarrow \textsc{Skip} \\
&&& \quad \textsc{Where}\ \neg(\mathit{dest} = \mathit{fl}) \\
&\mathit{Move2}(i, \mathit{dest}, \mathit{fl}) && \mathrel{\widehat{=}} \mathit{open}!i \rightarrow \mathit{confirm} \rightarrow \textsc{Skip} \\
&&& \quad \textsc{Where}\ \mathit{dest} = \mathit{fl} \\
&\mathit{Move}(i, \mathit{dest}, \mathit{fl}) && \mathrel{\widehat{=}} \mathit{Move1}(i, \mathit{dest}, \mathit{fl}) \Box \mathit{Move2}(i, \mathit{dest}, \mathit{fl}) \\
&\mathit{Get\_External}(i, \mathit{fl}) && \mathrel{\widehat{=}} \mathit{select}?(\mathit{dest}, \mathit{dir}) \rightarrow \mathit{Move}(i, \mathit{dest}, \mathit{fl}); \\
&&& \quad \mathit{service}!(\mathit{fl}, \mathit{dir}) \rightarrow \mathit{close} \rightarrow \textsc{Skip} \\
&\mathit{Get\_Internal}(i, \mathit{fl}, \mathit{md}) && \mathrel{\widehat{=}} \mathit{int\_sched}!(\mathit{fl}, \mathit{md}) \rightarrow \mathit{int\_sched}?(\mathit{dir}) \rightarrow \\
&&& \quad \mathit{check}!(\mathit{fl}, \mathit{dest}, \mathit{md}) \rightarrow \mathit{check} \rightarrow \mathit{Move}(i, \mathit{dest}, \mathit{fl}); \\
&&& \quad \mathit{init\_serv}!\mathit{fl} \rightarrow \mathit{close}?i \rightarrow \textsc{Skip} \\
&\mathit{LiftControl}(i, \mathit{fl}, \mathit{md}) && \mathrel{\widehat{=}} (\mathit{Get\_Internal}\mathit{fl}, \mathit{md}) \\
&&& \quad \Box(\textsc{Wait}[t_p];\ \mathit{Get\_External}(\mathit{fl}))); \\
&&& \quad \mathit{LiftControl}(i, \mathit{fl}, \mathit{md}) \\
&\mathit{Lift}(i) && \mathrel{\widehat{=}} \mathit{Door}(i) \parallel \mathit{Shalf}(i) \parallel \\
&&& \quad \mathit{LiftControl}(i, 0, 0) \parallel \mathit{Internal\_Q}(i, [])
\end{aligned}
$$

Appendix C

Complete Model of Smart Nursing Home

Patient Behaviors:

Patient status: {inRoom, inShowerRoom, outside, inBed}

$$
\begin{array}{ll}
Residence(i, status) & \widehat{=} (EnterRoom(i, status);\ Residence(i, inRoom)) \\
& \quad \Box\ (LeaveRoom(i, status);\ Residence(i, outside)) \\
& \quad \Box\ (Shower(i, status);\ Residence(i, status)) \\
& \quad \Box\ (BedActivity(i, status);\ Residence(i, status)) \\
& \quad \Box\ (Toilet(i, status);\ Residence(i, status)) \\
EnterRoom(i, status) & \widehat{=} [status == outside] enterRoom.i \rightarrow \textsc{Skip} \\
LeaveRoom(i, status)) & \widehat{=} [status == inRoom] leaveRoom.i \rightarrow \textsc{Skip} \\
BedActivity(i, status) & \widehat{=} [status == inRoom] gotoBed \rightarrow \\
& \quad ((sitonbed.0.i \rightarrow BedAct\_(i, 0)) \\
& \quad \Box\ (sitonbed.1.i \rightarrow BedAct\_(i, 1))) \\
BedAct\_(i, j) & \widehat{=} (SittingOnBed(i) \Box Sleeping(i)); \\
& \quad (BedAct\_(i, j) \Box GetUp(i, j)) \\
SittingOnBed(i) & \widehat{=} sitting.i \rightarrow \textsc{Skip} \\
Sleeping(i) & \widehat{=} laydown \rightarrow sleeping \rightarrow \textsc{Skip} \\
GetUp(i, j) & \widehat{=} leavebed.j.i \rightarrow \textsc{Skip} \\
Toilet(i, status) & \widehat{=} [status == inRoom]\ start\_toilet \stackrel{30}{\rightarrow} \\
& \quad (leave\_toilet \rightarrow \textsc{Skip} \Box flush \rightarrow leave\_toilet \rightarrow \textsc{Skip}); \\
& \quad (Skip \Box WashHand(i)) \\
WashHand(i, status) & \widehat{=} [status == inRoom]\ turnon_w ash\_tap \rightarrow \\
& \quad (washhand \stackrel{20}{\rightarrow} (turnoff\_wash\_tap \rightarrow fin\_washhand \rightarrow \textsc{Skip} \\
& \quad \Box\ fin\_washhand \rightarrow \textsc{Skip})) \\
Shower(i, status) & \widehat{=} [status == inRoom]\ enterShowerRoom.i \\
& \quad \rightarrow (Skip \Box Wash());\ leaveShowerRoom.i \rightarrow \textsc{Skip} \\
Wash() & \widehat{=} turnon\_shower\_tap \rightarrow start\_apply\_soap \stackrel{60}{\rightarrow} \\
& \quad (\textsc{Skip} \Box turnoff\_shower\_tap \rightarrow \textsc{Skip});\ getDressed \rightarrow \textsc{Skip}
\end{array}
$$

Sensor Groups:

$$
\begin{array}{ll}
BedPressureS(i, j) & \widehat{=} sitonbed.j.i \rightarrow bed\_pressure\_sensor!(j, 0) \rightarrow BedRFID(i, j) \\
& \quad \Box\ laydown.j.i \rightarrow bed\_pressure\_sensor!(j, 1) \rightarrow \textsc{Skip} \\
& \quad \Box\ leavebed.j.i \rightarrow BedRFID(-1, j) \\
& \quad \textsc{Where}\ sitonbed.j.i.\textsc{Engage} = \\
& \qquad\qquad bed\_pressure\_sensor!(j, 0).\textsc{Engage} \\
& \quad \wedge\ laydown.j.i.\textsc{Engage} = \\
& \qquad\qquad bed\_pressure\_sensor!(j, 1).\textsc{Engage} \\
BedPressureSensor & \widehat{=} (BedPressureS(0, 0) \Box BedPressureS(0, 1)) \ |||\ \\
& \quad (BedPressureS(1, 0) \Box BedPressureS(1, 1)); \\
& \quad BedPressureSensor \\
BedRFID(i, j) & \widehat{=} bed\_rfid!j.i \rightarrow \textsc{Skip}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{DoorRFIDReader} & \widehat{=} (\mathit{EnterDoorRFID} \,\Box\, \mathit{LeaveDoorRFID});\ \mathit{DoorRFIDReader} \\
\mathit{EnterDoorRFID} & \widehat{=} \mathit{enterRoom}.0 \rightarrow \mathit{rfid\_room}!(0, \mathit{outside}) \rightarrow \textsc{Skip} \\
& \quad \Box\ \mathit{enterRoom}.1 \rightarrow \mathit{rfid\_room}!(1, \mathit{outside}) \rightarrow \textsc{Skip} \\
& \quad \textsc{Where}\ \mathit{enterRoom}.0.\textsc{Engage} = \\
& \qquad\qquad \mathit{rfid\_room}!(0, \mathit{outside}).\textsc{Engage} \\
& \quad \wedge\ \mathit{enterRoom}.1.\textsc{Engage} = \mathit{rfid\_room}!(1, \mathit{outside}).\textsc{Engage} \\
\mathit{LeaveDoorRFID} & \widehat{=} \mathit{leaveRoom}.0 \rightarrow \mathit{rfid\_room}!(0, \mathit{inRoom}) \rightarrow \textsc{Skip} \\
& \quad \Box\ \mathit{leaveRoom}.1 \rightarrow \mathit{rfid\_room}!(1, \mathit{inRoom}) \rightarrow \textsc{Skip} \\
& \quad \textsc{Where}\ \mathit{leaveRoom}.0.\textsc{Engage} = \mathit{rfid\_room}!(0, \mathit{inRoom}) \\
& \quad \wedge\ \mathit{leaveRoom}.1.\textsc{Engage} = \mathit{rfid\_room}!(1, \mathit{inRoom}).\textsc{Engage} \\
\mathit{SRDoorRFIDReader} & \widehat{=} (\mathit{EnterSRRFID} \,\Box\, \mathit{LeaveSRRFID});\ \mathit{SRDoorRFIDReader} \\
\mathit{EnterSRRFID} & \widehat{=} \mathit{enterShowerRoom}.0 \rightarrow \mathit{rfid\_showerroom}!(0, \mathit{inRoom}) \rightarrow \textsc{Skip} \\
& \quad \Box\ \mathit{enterShowerRoom}.1 \rightarrow \mathit{rfid\_showerroom}!(1, \mathit{inRoom}) \\
& \qquad\qquad \rightarrow \textsc{Skip} \\
& \quad \textsc{Where}\ \mathit{enterShoerRoom}.0.\textsc{Engage} = \\
& \qquad\qquad \mathit{rfid\_showerroom}!(0, \mathit{inRoom}).\textsc{Engage} \\
& \quad \wedge\ \mathit{enterShowerRoom}.1.\textsc{Engage} = \\
& \qquad\qquad \mathit{rfid\_showerroom}!(1, \mathit{inRoom}).\textsc{Engage} \\
\mathit{LeaveSRRFID} & \widehat{=} \mathit{leaveShowerRoom}.0 \rightarrow \mathit{rfid\_showerroom}!(0, \mathit{inShowerRoom}) \\
& \qquad\qquad \rightarrow \textsc{Skip} \\
& \quad \Box\ \mathit{leaveShowerRoom}.1 \rightarrow \mathit{rfid\_showerroom}!(1, \mathit{inShowerRoom}) \\
& \qquad\qquad \rightarrow \textsc{Skip} \\
& \quad \textsc{Where}\ \mathit{leaveShowerRoom}.0.\textsc{Engage} = \\
& \qquad\qquad \mathit{rfid\_showerroom}!(0, \mathit{inShowerRoom}).\textsc{Engage} \\
& \quad \wedge\ \mathit{leaveShowerRoom}.1.\textsc{Engage} = \\
& \qquad\qquad \mathit{rfid\_showerroom}!(1, \mathit{inShowerRoom}).\textsc{Engage}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{ShowerSensor\_Tap} & \widehat{=}\ \mathit{turnon\_shower\_tap} \rightarrow \mathit{showerroom\_tap}!1 \rightarrow \textsc{Skip} \\
& \quad \Box\ \mathit{turnoff\_shower\_tap} \rightarrow \mathit{showerroom\_tap}!0 \rightarrow \textsc{Skip});\ \\
& \quad \mathit{ShowerSensor\_Tap} \\
& \quad \textsc{Where}\ \mathit{turnon\_shower\_tap}.\textsc{Engage} = \\
& \qquad\qquad \mathit{showerroom\_tap}!1.\textsc{Engage} \\
& \quad \wedge\ \mathit{turnoff\_shower\_tap}.\textsc{Engage} = \mathit{showerroom\_tap}!0.\textsc{Engage} \\
\mathit{WashSensor\_Tap} & \widehat{=} (\mathit{turnon\_wash\_tap} \rightarrow \mathit{washbasin\_tap}!1 \rightarrow \textsc{Skip} \\
& \quad \Box\ (\mathit{turnoff\_wash\_tap} \rightarrow \mathit{washbasin\_tap}!0 \rightarrow \textsc{Skip}); \\
& \quad \mathit{WashSensor}_{T}\mathit{ap}(); \\
& \quad \textsc{Where}\ \mathit{turnon}_{w}\mathit{ash\_tap}.\textsc{Engage} = \mathit{washbasin\_tap}!1.\textsc{Engage} \\
& \quad \wedge\ \mathit{turnoff\_wash\_tap}.\textsc{Engage} = \mathit{washbasin\_tap}!0.\textsc{Engage}
\end{array}
$$

$$
\begin{array}{ll}
\mathit{Sensors} \widehat{=} & \mathit{SRDoorRFIDReader} \,|||\, \mathit{ShowerSensor\_Tap} \,|||\, \mathit{BedPressureSensor} \\
& |||\, \mathit{DoorRFIDReader} \,|||\, \mathit{WashSensor\_Tap}
\end{array}
$$

$$
\begin{aligned}
LocationMonitor = \widehat{=}\ & (rfid\_room?(i, status) \rightarrow \\
& [status == outside]detectEnterRoom.i \rightarrow \text{SKIP} \\
& \Box\ [status == inRoom]detectLeaveRoom.i \rightarrow \text{SKIP}); \\
& \qquad LocationMonitor \\
ShowerRoomRFID\ \widehat{=}\ & (rfid\_showerroom?(i, status) \rightarrow \\
& [status == inRoom]detectEnterShowerRoom.i \rightarrow \text{SKIP} \\
& \Box\ [status == inShowerRoom]detectLeaveShowerRoom.i \\
& \qquad \rightarrow tapoff\_prompt \rightarrow \text{SKIP});\ ShowerRoomRFID \\
ShowerTapMonitor\ \widehat{=}\ & (showerroom\_tap?1 \rightarrow tapIsOn \rightarrow \text{SKIP} \\
& \Box\ showerroom\_tap?0 \rightarrow tapIsOff \rightarrow \text{SKIP}); \\
& \qquad ShowerTapMonitor \\
WashTapMonitor\ \widehat{=}\ & \\
& (washbasin\_tap?1 \rightarrow basinTapIsOn \rightarrow \text{SKIP} \\
& \Box\ washbasin\_tap?0 \rightarrow basinTapIsOff \rightarrow \text{SKIP}); \\
& \qquad WashTapMonitor \\
BedMonitor\ \widehat{=}\ & (bed\_pressure_sensor?(j, s) \rightarrow bed\_rfid?(i, j) \rightarrow \\
& [i! = -1 \wedge i! = j]prompt\_sleepInWrongBed.i \rightarrow \text{SKIP} \\
& \Box\ [i! = -1 \wedge i == j]prompt\_goodnight.i \rightarrow \text{SKIP}); \\
& \qquad BedMonitor \\
Monitor\ \widehat{=}\ & LocationMonitor \\
& |||\ ShowerRoomRFID \\
& |||\ ShowerTapMonitor \\
& |||\ BedMonitor \\
& |||\ WashTapMonitor
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{Reminder\_Toilet}(\mathit{duration}) \;\widehat{=}\; (\mathit{start\_toilet} \rightarrow \textsc{Skip} \overset{\mathit{duration}}{\triangleright} \\
&\quad \mathit{finish\_toilet} \rightarrow \mathit{prompt}!\mathit{toilet} \rightarrow \textsc{Skip}); \\
&\quad \mathit{Reminder\_Toilet}(\mathit{duration}) \\
&\mathit{Reminder\_Flushing}(\mathit{duration}) \;\widehat{=}\; (\mathit{start\_toilet} \rightarrow \mathit{finish\_toilet} \rightarrow \textsc{Skip} \overset{\mathit{duration}}{\triangleright} \\
&\quad \mathit{flush} \rightarrow \mathit{prompt}_{\mathit{f}}\mathit{lushtoilet} \rightarrow \textsc{Skip}); \\
&\quad \mathit{Reminder\_Flushing}(\mathit{duration}) \\
&\mathit{Reminder\_ShowerLong}(\mathit{duration}) \;\widehat{=}\; (\mathit{detectEnterShowerRoom}.0 \rightarrow \textsc{Skip} \overset{\mathit{duration}}{\triangleright} \\
&\quad \mathit{detectLeaveShowerRoom}.0 \\
&\qquad \rightarrow \mathit{prompt}!\mathit{showertoolong} \rightarrow \textsc{Skip} \\
&\quad \Box\ \mathit{detectEnterShowerRoom}.1 \rightarrow \textsc{Skip} \overset{\mathit{duration}}{\triangleright} \\
&\quad \mathit{detectLeaveShowerRoom}.1 \\
&\qquad \rightarrow \mathit{prompt}!\mathit{showertoolong} \rightarrow \textsc{Skip}); \\
&\quad \mathit{Reminder\_ShowerLong}(\mathit{duration}) \\
&\mathit{Reminder\_SleepingInWrongBed} \;\widehat{=}\; (\mathit{prompt\_sleepInWrongBed}.0 \\
&\qquad \rightarrow \mathit{prompt}!(\mathit{wrong\_bed}, 0) \rightarrow \textsc{Skip} \\
&\quad \Box\ \mathit{prompt\_sleepInWrongBed}.1 \\
&\qquad \rightarrow \mathit{prompt}!(\mathit{wrong\_bed}, 1) \rightarrow \textsc{Skip}); \\
&\quad \mathit{Reminder\_SleepingInWrongBed} \\
&\mathit{Prompting} \;\widehat{=}\; \mathit{prompt}?\mathit{status} \rightarrow \mathit{prompting} \rightarrow \mathit{Prompting}
\end{aligned}
$$

Appendix D

Complete PAT Model of Smart Nursing Home

```
#define N 2; //No. of residents, 2 residents per room
#define Bed 2; // No. of beds

//channels
channel bed_pressure_sensor_0 0; //pressure sensor for bed 0;
channel bed_pressure_sensor_1 0; // pressure sensor for bed 1;
channel chair_pressure_sensor 0 ;
channel rfid_room 0;
channel bed_rfid_0 0;
channel bed_rfid_1 0;

//channels
channel showerroom_reed 0;
channel rfid_showerroom 0;
channel showerroom_tap  0;
channel washbasin_tap 0;
channel doorlock 0;
channel shakesensor 0;
channel reminderc 0;

/* status of residences */
var isInRoom[2] = [1,1];
var isShowering[2] = [0, 0];
var bedStatus[2] = [-1, -1];
var isInBed[2] = [-1, -1];
var isInShowerRoom = [0, 0];

/* tap status */
var isShowerTapOn = false;
var isBasinTapOn = false;
```

$$
\begin{aligned}
&// - - - - - - - - - \ ADLS \ - - - - - - - - - \\
&Residence(i) = (Shower(i) \ \Box \ Sleep(i) \ \Box \ EnterRoom(i) \\
&\qquad \Box \ LeaveRoom(i) \ \Box \ Toilet(i)) \ ; \ Residence(i); \\
&Sleep(i) = \ if(isInRoom[i] == 1) \ \{gotoBed \rightarrow \\
&\qquad (((sitonbed\_0.i \rightarrow SittingOnBed(i); \\
&\qquad (Sleeping(i) \ \Box \ Skip); \ GetUp(i, 0)) \\
&\qquad \Box \ (sitonbed\_1.i \rightarrow SittingOnBed(i); \\
&\qquad (Sleeping(i) \ \Box \ Skip); \ GetUp(i, 1)))\}; \\
&SittingOnBed(i) = sitting.i \rightarrow Skip; \\
&Sleeping(i) = laydown \rightarrow sleeping \rightarrow Skip; \\
&GetUp(i, j) = if(j == 0)\{leavesbed\_0.i \rightarrow Skip\} \\
&\qquad else\{leavesbed\_1.i \rightarrow Skip\}; \\
&EnterRoom(i) = ifb(isInRoom[i] == 0)\{enterRoom.i \rightarrow Skip\}; \\
&LeaveRoom(i) = ifb(isInRoom[i] == 1 \ \&\& \ isInBed[i] \ != \ 1 \\
&\qquad \&\& \ isShowering[i] == 0 \ )\{leaveRoom.i \rightarrow Skip\}; \\
&Toilet(i) = if(isInRoom[i] == 1) \ \{start\_toilet \rightarrow Wait[30]; \\
&\qquad (leave\_toilet \rightarrow Skip \ \Box \ flush \rightarrow leave\_toilet \rightarrow Skip); \\
&\qquad (Skip \ \Box \ WashHand(i))\}; \\
&WashHand(i) = if(isInRoom[i] == 1) \ \{turnon\_wash\_tap \rightarrow \\
&\qquad (washhand \rightarrow Wait[20]; \ (turnoff\_wash\_tap \rightarrow \\
&\qquad fin\_washhand \rightarrow Skip \ \Box \ fin\_washhand \rightarrow Skip))\}; \\
&Shower(i) = readyForShower \rightarrow Wait[30]; \\
&\qquad if(isInRoom[i] == 1) \ \{ \\
&\qquad enterShowerRoom.i \rightarrow (Skip \ \Box \ Wash()); \\
&\qquad leaveShowerRoom.i \rightarrow Skip\}; \\
&Wash() = turnon\_shower\_tap \rightarrow start\_apply\_soap \rightarrow Wait[60, \ 2000]; \\
&\qquad (Skip \ \Box \ turnoff\_shower\_tap \rightarrow Skip); \\
&\qquad getDressed \rightarrow Skip;
\end{aligned}
$$

$$
\begin{aligned}
&/* \ ---------- \ Sensors \ ---------- */\\
&BedPressureSensor() =\\
&\quad ((sitonbed\_0.0 \rightarrow bed\_pressure\_sensor\_0!0 \rightarrow BedRFID(0,0))\\
&\quad \Box\ (sitonbed\_0.1 \rightarrow bed\_pressure\_sensor\_0!0 \rightarrow BedRFID(1,0))\\
&\quad \Box\ (sitonbed\_1.0 \rightarrow bed\_pressure\_sensor\_1!0 \rightarrow BedRFID(0,1))\\
&\quad \Box\ (sitonbed\_1.1 \rightarrow bed\_pressure\_sensor\_1!0 \rightarrow BedRFID(1,1))\\
&\quad \Box\ (laydown \rightarrow bed\_pressure\_sensor\_0!1 \rightarrow Skip)\\
&\quad \Box\ (laydown \rightarrow bed\_pressure\_sensor\_1!1 \rightarrow Skip)\\
&\quad \Box\ (leavesbed\_0.0 \rightarrow BedRFID(-1,0))\\
&\quad \Box\ (leavesbed\_0.1 \rightarrow BedRFID(-1,0))\\
&\quad \Box\ (leavesbed\_1.0 \rightarrow BedRFID(-1,1))\\
&\quad \Box\ (leavesbed\_1.1 \rightarrow BedRFID(-1,1))\\
&\quad); \ BedPressureSensor();\\
&BedRFID(i,j) = if(j == 0)\ \{bed\_rfid\_0!i \rightarrow Skip\}\\
&\quad else\ \{bed\_rfid\_1!i \rightarrow Skip\};\\
&DoorRFIDReader() = (\ atomic\{enterRoom.0 \rightarrow rfid\_room!0 \rightarrow Skip\}\\
&\quad \Box\ atomic\{enterRoom.1 \rightarrow rfid\_room!1 \rightarrow Skip\}\\
&\quad \Box\ atomic\{leaveRoom.0 \rightarrow rfid\_room!0 \rightarrow Skip\}\\
&\quad \Box\ atomic\{leaveRoom.1 \rightarrow rfid\_room!1 \rightarrow Skip\}\\
&\quad); \ DoorRFIDReader();\\
&ShowerRoomDoorRFIDReader() =\\
&\quad (atomic\{enterShowerRoom.0 \rightarrow rfid\_showerroom!0 \rightarrow Skip\}\\
&\quad \Box\ atomic\{enterShowerRoom.1 \rightarrow rfid\_showerroom!1 \rightarrow Skip\}\\
&\quad \Box\ atomic\{leaveShowerRoom.0 \rightarrow rfid\_showerroom!0 \rightarrow Skip\}\\
&\quad \Box\ atomic\{leaveShowerRoom.1 \rightarrow rfid\_showerroom!1 \rightarrow Skip\}\\
&\quad); \ ShowerRoomDoorRFIDReader();\\
&ShowerSensor\_Tap() =\\
&\quad \Box\ i : \{0..N-1\}@(atomic\{turnon\_shower\_tap\\
&\quad \rightarrow showerroom\_tap!1 \rightarrow Skip\}\\
&\quad \Box\ (atomic\{turnoff\_shower\_tap \rightarrow showerroom\_tap!0 \rightarrow Skip\})\\
&\quad); \ ShowerSensor\_Tap();\\
&WashSensor\_Tap() = \Box\ \ i : \{0..N-1\}@((\\
&\quad atomic\{turnon\_wash\_tap \rightarrow washbasin\_tap!1 \rightarrow Skip\ \})\\
&\quad \Box\ (atomic\{turnoff\_wash\_tap \rightarrow washbasin\_tap!0 \rightarrow Skip\ \})\\
&\quad); \ WashSensor\_Tap();\\
&Sensors() = ShowerRoomDoorRFIDReader()\\
&\quad |||\ \ ShowerSensor\_Tap()\\
&\quad |||\ \ BedPressureSensor()\\
&\quad |||\ \ DoorRFIDReader()\\
&\quad |||\ \ WashSensor\_Tap();
\end{aligned}
$$

$$
\begin{aligned}
&/* \ ---------- \textit{Monitor System} ---------- \\
&\textit{LocationMonitor}() = \textit{atomic}\{\textit{rfid\_room}?i \rightarrow \\
&\quad \textit{if}(\textit{isInRoom}[i] == 0)\{\textit{tau}\{\textit{isInRoom}[i] = 1;\ \} \rightarrow \textit{Skip}\} \\
&\quad \textit{else}\{ \\
&\quad \textit{if}(\textit{isInRoom}[i] == 1)\{\ \textit{tau}\{\textit{isInRoom}[i] = 0;\ \} \rightarrow \textit{Skip}\} \\
&\quad \}\};\ \textit{LocationMonitor}(); \\
&\textit{ShowerRoomRFID}() = \textit{atomic}\{\textit{rfid\_showerroom}?i \rightarrow \\
&\quad \textit{if}(\textit{isInShowerRoom}[i] == 0)\ \{ \\
&\quad \textit{tau}\{\textit{isInShowerRoom}[i] = 1;\ \} \rightarrow \textit{Skip}\} \\
&\quad \textit{else}\{ \\
&\quad \textit{if}(\textit{isInShowerRoom}[i] == 1)\{ \\
&\quad \textit{tau}\{\textit{isInShowerRoom}[i] = 0;\ \} \rightarrow \textit{tapoff\_prompt} \rightarrow \textit{Skip}\ \}\} \\
&\quad \};\ \textit{ShowerRoomRFID}(); \\
&\textit{ShowerTapMonitor}() = (\textit{atomic}\{\textit{showerroom\_tap}?1 \rightarrow \\
&\quad \textit{tau}\{\textit{isShowerTapOn} = \textit{true};\ \} \rightarrow \textit{Skip}\} \\
&\quad \Box\ \ \textit{atomic}\{\textit{showerroom\_tap}?0 \rightarrow \textit{tau}\{\textit{isShowerTapOn} = \textit{false};\ \} \rightarrow \\
&\quad \textit{Skip}\});\ \textit{ShowerTapMonitor}(); \\
&\textit{WashTapMonitor}() = (\textit{atomic}\{\textit{washbasin\_tap}?1 \\
&\quad \rightarrow \textit{tau}\{\textit{isBasinTapOn} = \textit{true};\ \} \rightarrow \textit{Skip}\} \\
&\quad \Box\ \ \textit{atomic}\{\textit{washbasin\_tap}?0 \rightarrow \textit{tau}\{\textit{isBasinTapOn} = \textit{false};\ \} \\
&\quad \rightarrow \textit{Skip}\})\ ;\ \textit{WashTapMonitor}(); \\
&\textit{BedMonitor}() = \textit{BedMonitor}1() \ |||\ \textit{BedMonitor}2(); \\
&\textit{BedMonitor}1() = (\textit{atomic}\{\textit{bed\_pressure\_sensor\_0}?i \rightarrow \textit{tau}\{\textit{isInBed}[0] = i;\ \} \rightarrow \\
&\quad \textit{if}(i! = -1\ \&\&\ i! = 0)\ \{\textit{sleepInWrongBed}.1 \rightarrow \textit{Skip}\}\} \\
&\quad \Box\ \ \textit{atomic}\{\textit{bed\_pressure\_sensor\_1}?i \rightarrow \textit{tau}\{\textit{isInBed}[1] = i;\ \} \rightarrow \\
&\quad \textit{if}(i! = -1\ \&\&\ i! = 1)\ \{\textit{sleepInWrongBed}.0 \rightarrow \textit{Skip}\}\}); \\
&\quad \textit{BedMonitor}1(); \\
&\textit{BedMonitor}2() = \ (\textit{atomic}\{\textit{bed\_rfid\_0}?i \rightarrow \textit{tau}\{\textit{bedStatus}[0] = i;\ \} \rightarrow \textit{Skip}\} \\
&\quad \Box\ \ \textit{atomic}\{\textit{bed\_rfid\_1}?i \rightarrow \textit{tau}\{\textit{bedStatus}[1] = i;\ \} \rightarrow \textit{Skip}\}); \\
&\quad \textit{BedMonitor}2(); \\
&\textit{Monitor}() = \textit{LocationMonitor}() \\
&\quad |||\ \ \textit{ShowerRoomRFID}() \\
&\quad |||\ \ \textit{ShowerTapMonitor}() \\
&\quad |||\ \ \textit{BedMonitor}() \\
&\quad |||\ \ \textit{WashTapMonitor}();
\end{aligned}
$$

$$
\begin{aligned}
&/* \ ---------- \textit{Services/Reminders/Control System} \ ------ \\
&\textit{Toilet\_Reminder}(\textit{duration}) = \\
&\qquad \textit{start\_toilet} \rightarrow \textit{Skip timeout}[\textit{duration}]\ \textit{finish\_toilet} \rightarrow \\
&\qquad \textit{prompt\_toilet} \rightarrow \textit{Skip};\ \textit{Toilet\_Reminder}(\textit{duration}); \\
&\textit{Flush\_Toilet\_Reminder}(\textit{duration}) = \\
&\qquad \textit{start\_toilet} \rightarrow \textit{Skip timeout}[\textit{duration}]\ \textit{flush} \rightarrow \\
&\qquad \textit{prompt\_flushtoilet} \rightarrow \textit{Skip}; \\
&\textit{Reminder\_Shower\_TooLong}(\textit{duration}) = \\
&\qquad \Box\ \ i : \{0..N-1\}@\textit{enterShowerRoom}.i \rightarrow (\textit{leaveShowerRoom}.i \\
&\qquad\ \ \rightarrow \textit{Skiptimeout}[\textit{duration}]\ \textit{prompt\_finishshower} \rightarrow \textit{Skip}); \\
&\textit{SleepingInWrongBedReminder}() = \\
&\qquad \Box\ \ i : \{0..N-1\}@\textit{atomic}\{\textit{sleepInWrongBed}.i \rightarrow \\
&\qquad \textit{prompt\_wrong\_bed}.i \rightarrow \textit{Skip}\};\ \textit{SleepingInWrongBedReminder}(); \\
&\textit{Reminder\_TurnOff\_Tap}() = \textit{atomic}\{\textit{tapoff\_prompt} \rightarrow \textit{if}(\textit{isShowerTapOn} == \textit{true}) \\
&\qquad \{\textit{if}(\textit{isInShowerRoom}[0] == 0\ \&\&\ \textit{isInShowerRoom}[1] == 0\ ) \\
&\qquad \{\textit{tapon\_nobodyin} \rightarrow \textit{Skip}\};\ \textit{prompt\_TurnOffTap} \rightarrow \\
&\qquad \textit{CheckFalseAlarm}(1)\ \}\};\ \textit{Reminder\_TurnOff\_Tap}(); \\
&\\
&/* \ ---------- \textit{Errorness cases} \ ---------- \\
&\textit{var falsealarm} = \textit{false}; \\
&\textit{CheckFalseAlarm}(\textit{type}) = \textit{if}(\textit{type} == 1)\{\textit{if}(\textit{isInShowerRoom}[0] == 1 \\
&\qquad\ \ \| \ \textit{isInShowerRoom}[1] == 1)\{\textit{tau}\{\textit{falsealarm} = \textit{true};\ \} \rightarrow \textit{Skip}\}\}; \\
&\\
&/* \ ---------- \textit{Define Smart Nursing Home System} \ ---------- \\
&\textit{Reminders}() = \textit{Toilet\_Reminder}(1800)\ |||\ \textit{Reminder\_Shower\_TooLong}(1800) \\
&\quad |||\ \textit{SleepingInWrongBedReminder}()\ |||\ \textit{Reminder\_TurnOff\_Tap}()\ ; \\
&\textit{Residences}() = \textit{Residence}(0)\ |||\ \textit{Residence}(1)\ ; \\
&\textit{SmartNursingHome}() = (\textit{Residences}()\ \|\ \textit{Sensors}()\ \|\ \textit{Reminders}())\ \ \|\ \textit{Monitor}()\ ;
\end{aligned}
$$