Contents

1 Introduction **5**

2 Related work **9**

2.1 Modelling formalisms 9

2.1.1 Timed Automata 10

2.1.2 Petri Nets 10

2.1.3 Timed Process Algebras 11

2.1.4 Timed Temporal Logics 11

2.2 Implementation language 13

2.3 Model-checkers 13

2.3.1 Kronos 14

2.3.2 UPPAAL 14

2.3.3 RT-Spin 14

2.3.4 Rabbit 15

2.3.5 MOCHA 15

2.3.6 Java PathFinder 16

3 Case Study **17**

3.1 Fischer Protocol 19

3.1.1 RTSJ Implementation 20

3.1.2 UPPAAL 21
3.1.3 RT-Spin 24
3.2 Conclusion 26

4 A Non-Trivial example: A Gear Controller 29

4.1 Gear Controller 30
4.2 Model-based approach 31
4.2.1 Overview 32
4.2.2 Mapping details 35
4.2.3 Verification result 39
4.3 From RTSJ to RT-Promela 41
4.3.1 Overview 41
4.3.2 Mapping details 42
4.3.3 Verification result 44
4.4 Conclusion 45

5 Remainder of the project 47

5.1 Timetable 50

List of Figures

3.1 Fischer Protocol Pseudo-code 20
3.2 RT-Java code modelling Fischer Protocol 22
3.3 Fischer protocol modelled in UPPAAL 23
3.4 Mutex query and the result of its verification 24
3.5 Fischer Protocol modelled in RT-Promela 25
3.6 Mutex query 26

4.1 Timed Automaton representing Clutch 31
4.2 Architecture 32
4.3 RTSJ code modelling Clutch 33
4.4 Architecture 41
4.5 Promela code modelling Clutch 43

5.1 Research Timetable 51

Chapter 1

Introduction

Real-time [1] is a broad term used to describe applications that have timing requirements. In other words, real-time systems are software systems in which the correctness of the system depends not only on the logical results of computations, but also on the time it takes to produce those results. It is common to make a distinction between hard and soft real-time systems [2]. Hard real-time systems are systems that must always react within the specified deadline. On the other hand, soft real-time systems may occasionally fail to meet their timing requirements. Soft real-time systems can be distinguished from interactive systems in which there are no explicit deadlines [2]. As an example, the remote control system of a car engine is a hard real-time system because a delayed signal may cause engine failure or damage, whereas a data acquisition system for a process control application is soft.

Many safety- and security-critical systems are hard real-time systems and, as a result, tools and techniques for verifying real-time systems are extremely important. The traditional ways of ensuring that real-time systems operate correctly have been simulation and testing. However, in many cases these techniques are exceedingly time-consuming and provide only probabilistic

measures of correctness. Formal methods advocate the use of mathematical reasoning as an alternative; one of the most promising of these methods has been model-checking [3]. Model-checking is a state-exploration technique to decide if a property is satisfied in a finite state machine [4]. A property describes a particular feature, often expressed as a temporal logic formula [5], that is a partial specification of the intended behaviour. The property is automatically compared with a state-transition graph representing the behaviour of the system.

The Timed Automata [6] formalism has been used to model and verify real-time systems. A Timed Automaton is a finite-state automaton extended with a finite set of real-valued variables modelling clocks. In the last decade, there have been a number of tools developed based on Timed Automata to model and verify real-time systems, notably Kronos [7], UPPAAL [8], RT-Spin [9], Rabbit [10] and MOCHA [11]. However, these tools provide formal verification for models, not programs, which means it is still necessary to show that the programs that implement those models satisfy the properties as well.

Real-time systems have to generate their output within a finite and predictable time. Therefore, the specification of the language in which real-time systems are implemented is important. Real-Time Specification of Java (RTSJ) [2] was proposed in January 2002. Sun has developed a simulator, Java Real-Time System (Java RTS) 2.0 [12], for simulating real-time Java code that is compliant with the RTSJ.

To verify real-time Java code that is compliant with the RTSJ, an approach based on JPF(Java Path Finder) [13] has been proposed by Lindstrom et al. [14]. JPF is a Java model-checker which has a state-exploring JVM (Java Virtual Machine) at its core — JPF explores every JVM state instead

of just considering a single path, which is what other JVM implementations do. However, the approach based on JPF to verify real-time Java code has not been implemented yet, and it only supports properties that are specified as normal Java assertions, without timing constraints.

Increasing the confidence of real-time programs written in RTSJ is the main topic of this project. In this research, a model-based approach will be proposed for verifying such systems. A real-time model is a simplified representation of a real-time system. Models focus on system behaviour and abstract many details of programs [15]. More importantly, these models can be verified mechanically with real-time model-checkers.

As an initial task, formalisms for modelling timing systems and a number of model-checkers were studied. As a result of this study, UPPAAL and RT-Spin were selected as the real-time model checking tools to investigate more closely. Then, to increase our understanding of RTSJ, and also to become familiar with Java RTS, UPPAAL and RT-Spin, a typical real-time example called the Fischer protocol [16] was implemented, simulated and verified with RTS, RT-Spin and UPPAAL. Next, by means of a non-trivial example, a gear controller [17], two methods for model-checking RTSJ programs were investigated:

- Using a model-based approach to derive RTSJ code from Timed Automata supported by UPPAAL.
- Using a mapping algorithm to map RTSJ code to RT-Promela supported by RT-Spin.

Based on the results of this phase of the project, we decided to pursue the model-based approach using UPPAAL and RTSJ in the remainder of the

research. This part of the project was peresented at the Verify'08 conference [18].

The main contribution of this work will be a model-based approach based on Timed Automata, the UPPAAL model checker and RTSJ. First, models can be mechanically verified, to check whether they satisfy particular properties, by using UPPAAL. Then, programs are derived from the model by following a systematic approach. This transformation will be partially automated and the mapping will be proved semi-formally. As not all features can be translated in such a way that properties are preserved and not all properties can be preserved, the mapping is expected to be partial. The properties will be divided into three categories:

- Properties that are guaranteed to hold unconditionally in the code when they hold in the model.
- Properties that are guaranteed to hold provided that certain conditions are specified by the RTSJ code.
- Properties that cannot be guaranteed to hold based on the mapping. For checking these properties, alternative V&V techniques will be suggested.

Chapter 2 briefly reviews formalisms and languages that have been proposed for modelling real-time systems. In this chapter, related work on real-time model-checking, JPF and RTSJ are also discussed. A typical real-time example called the Fischer protocol is implemented, simulated and verified with RTS, RT-Spin and UPPAAL, in Chapter 3. Chapter 4 discusses approaches for model-checking RTSJ with UPPAAL and RT-Spin by means of the gear controller example. The plan for the remainder of the project are described in Chapter 5.

Chapter 2

Related work

2.1 Modelling formalisms

Traditional formalisms for temporal reasoning deal with the qualitative aspect of time, that is, the order of certain system events (an example of a qualitative time property is: event A occurs before event B). However, real-time systems often require quantitative aspects of time. This means they need to consider the actual difference in time between certain system events.

Timed Automata is a formalism for the modelling and verification of real-time systems. Examples of other formalisms are Timed Petri Nets [19], Time Petri Nets [20], Timed Process Algebras [21], and Real-time Logics [22].

In this section, we provide a brief overview of these modelling formalisms which are used for specifying and verifying real-time systems. Timed Automata and Real-time Logics are described in more detail, as they are the formalisms behind the real-time model-checkers that we discuss in the next section.

2.1.1 Timed Automata

A Timed Automaton is a finite-state automaton extended with a finite set of real-valued variables modelling clocks. Timed words in a Timed Automaton are infinite sequences in which a real-valued time of occurrence is associated with each symbol [6]. Each clock can be reset to zero with the transitions of the automaton, and keeps track of the time elapsed since the last reset. The transitions of the automaton put certain constraints on the clock values. A transition may be taken only if the current values of the clocks satisfy the associated constraints. To quote Alur and Dill [23]:

> Timed Automata can capture several interesting aspects of real-time systems: qualitative features such as liveness, fairness, and nondeterminism; and quantitative features such as periodicity, bounded response, and timing delays.

More formally, a Timed Automaton is a tuple $A = (S, \Sigma, Tr, S_0, F, C)$, where S is a finite set of states, Σ an alphabet, Tr a transition relation, S_0 an initial state, F a set of accepting or final states, and C a finite set of clocks [9].

2.1.2 Petri Nets

Two Petri Net based models for handling time have been proposed: Timed Petri Nets [20] and Time Petri Nets [19].

Timed Petri Nets are derived from Petri Nets by associating a finite firing duration with each transition of the net. The new firing rule of Timed Petri Nets differs from the classical firing rule of Petri Nets in that it accounts for the time it takes to fire a transition and a transition must fire as soon as it is enabled. These nets have been used mainly for performance evaluation.

Time Petri Nets differ from Timed Petri Nets in that they have two real numbers, representing upper and lower bounds on the time it takes to fire a transition, instead of having only one firing time as is the case in Timed Petri Nets [24, 20].

2.1.3 Timed Process Algebras

Timed Process Algebras are classified based on whether they use absolute or relative time, and whether they use time progression constructs or time stamping of actions [25].

μCRL (micro common representation language) is a process algebra language mostly used for specification and verification of communication protocols. A timed version of the μCRL language, called $Timed\mu CRL$, has been proposed by Groote [26]. Time is an absolute and abstract data type in $Timed\mu CRL$ satisfying various conditions and with a construct to ensure an action happens at a specific time [27]. Other examples of Timed Process Algebras are provided by Baeten [25].

2.1.4 Timed Temporal Logics

In Timed Temporal Logics, system behaviour can be specified in terms of logical formulas, including temporal constraints, events and the relationships between the two by temporal operators.

Temporal logic is a tool for reasoning about graphs [28] and was originally developed to investigate logic under the modes of *necessary* and *possible* truth. Temporal logic is closely linked to both first- and second-order classical logic. The relationship between Temporal logic and other forms of logic is discussed by Blackburn and Van Benthem [29]. Temporal logic plays an important role in the specification, derivation, and verification of programs

which can be viewed as progressing through a sequence of states, a new state after each event in the system. They have a particularly useful role in the specification and verification of communication protocols and reactive systems.

Temporal logic model checking involves checking the state-space of a model of a system to determine whether errors can occur in the system (the system can transit from its current state to its immediate successor state if the constraints on the transition edge hold) [30]. Quantitative temporal relationships can be defined in Timed Temporal Logics. In specifying real-time systems, the general behaviour of the system is typically expressed by means of quantitative temporal constraints. The correct behaviour of the system depends on whether it satisfies these temporal constraints.

The Computational Tree Logic (CTL) presented by Clarke et al. is a propositional branching time temporal logic [31]. Consider P as a set of atomic propositions. CTL formulas are defined inductively as follows:

$$\Phi := p|false|\Phi_1 \rightarrow \Phi_2|\exists o\Phi_1|\exists\Phi_1\mu\phi_2|\forall\Phi_1\mu\phi_2$$

where $p \in P$, and Φ_1 and Φ_2 are CTL-formulas. $\exists o\Phi_1$ expresses that there is an immediate successor state in which Φ_1 holds; in other words, by executing exactly one step Φ_1 holds. $\exists\Phi_1\mu\phi_2$ expresses that for some computation path, there exists an initial prefix of the path such that Φ_2 holds in the last state of the prefix and Φ_1 holds in all the intermediate states. $\forall\Phi_1\mu\Phi_2$ expresses that for every computation path, the above property holds. A timed extension of CTL is TCTL (Timed Computational Tree Logic) [32]. TCTL provides a way to put a bound on the time at which an atomic property will become true, by permitting the definition of a bounded operator (e.g. $\forall_{[6,9]}\exists E$ expresses that E will be true sometimes from 6 to 9 time units from the current time) [5].

2.2 Implementation language

Real Time Specification of Java (RTSJ) [2] was proposed in January 2002. RTSJ is designed to support both hard and soft real-time applications. RTSJ adds several features to Java such as Clocks, Time, Scoped Memory Areas which provide guarantees on allocation time, Fixed Priority Scheduling Policy, Asynchronous Events and Real-Time Threads. To remove uncertainties of the traditional Java GC (Garbage Collection), RTSJ also provides an approach for memory management in which time that is consumed by GC is divided into a series of increments called quanta and a percentage of time in a given window of time is allocated for these quanta.

To support the simulation of real-time Java code, Sun has developed a simulator, Java Real-Time System (Java RTS) 2.0. This release is compliant with the Real-Time Specification for Java (RTSJ) [12].

2.3 Model-checkers

This section provide an overview of existing tools for model checking real-time systems and the RTSJ.

Model-checking of timed automata representations has become popular for the analysis of real-time systems [33, 6]. In the last decade, there have been a number of tools developed based on timed automata to model and verify real-time systems, notably Kronos [7], UPPAAL [8], RT-Spin [9], Rabbit [10] and MOCHA [11]. On the other hand, JPF is a Java model-checker; an approach to verify real-time Java code with JPF has been proposed by Lindstrom et al. [14].

2.3.1 Kronos

Kronos is a real-time verification tool [7]. Kronos is based on the formalisms of Timed Automata and Timed Temporal Logics. Predicates on the values of the clocks express timing constraints that can be propagation delays, execution times and response times. To model inter-process communication, transitions of the timed automata are labelled with sets of identifiers interpreted as synchronisation channels. Kronos can check properties expressed in both TCTL (logical approach) and Timed Automata (behavioural approach).

2.3.2 UPPAAL

UPPAAL is a tool-box for modelling, simulation and verification of real-time systems [8]. The core of UPPAAL is a modelling language consisting of networks of Timed Automata. The language has been extended with features such as shared integer variables and urgent channels. UPPAAL consists of three parts: a description language, a simulator and a model-checker. The model-checking engine of UPPAAL is designed to check a subset of TCTL formulas for networks of Timed Automata.

2.3.3 RT-Spin

RT-Spin is a tool for the verification and simulation of RT-Promela [34]. Promela [34] is a C-like language that is supported by the Spin model-checker. RT-Promela is a real-time extension of the Promela language in which clock variables are declared globally. Clock variables can be reset independently. Each statement is expanded with an optional time part. The mathematical foundation of RT-Spin is the formalism of Timed Automata [9]. RT-Spin accepts properties expressed by Linear Temporal logic (LTL) [35],

which are translated to Promela code. The syntax for LTL is similar to the syntax of CTL. The main difference between LTL and CTL formulas is the absence of $\exists$ (existential)operator in LTL formulas which reflects the linear-time paradigm.

2.3.4 Rabbit

Another tool for model-checking of Timed Automata is Rabbit [10]. An extension of Timed Automata called Cottbus Timed Automata is used by Rabbit. Automata describing the behaviour of the system are encapsulated by modules and communication with other modules is through shared variables and synchronisation labels declared within an interface.

2.3.5 MOCHA

MOCHA is an interactive verification environment for modular verification of heterogeneous systems [11]. The properties in MOCHA are described by ATL(Alternating Temporal Logic). ATL is a Temporal Logic that is designed to specify requirements of open systems. ATL specifies collaboration as well as interactions between components. An ATL model-checking algorithm has been proposed that is similar to that of CTL model-checking [36].

MOCHA has a real-time system verifier and three modes of module simulation: random simulation, manual simulation and game simulation (some of the modules are executed by the simulator, while the remaining ones are executed by the user).

2.3.6 Java PathFinder

The Java model-checker Java Path Finder(JPF) [14] has a state-exploring JVM (Java Virtual Machine) at its core. JPF examines alternative paths in a Java program (via backtracking). JPF uses discrete event simulation as a basis for modelling time. There is a proposal for verifying real-time Java with JPF [14]. However, this has not yet been implemented.

Moreover, only non-timing properties are supported by the JPF proposal, to quote Lindstrom et al. [14]:

> JPF's approach differs from the other approaches used in Kronos, Rabbit and UPPAAL in that it analyses systems with complex transitions but with simple explicit timing information, whereas in the Timed Automata approach systems are analysed with complex timing, but simple transitions (e.g., between abstract states in given time intervals).

Chapter 3

Case Study

To obtain confidence that programs written in RTSJ satisfy certain properties, it is essential to have an adequate understanding of RTSJ features and various ways of describing properties in real-time Java programs. Therefore, as an initial task, a feasibility study was carried out. The main purpose of this feasibility study was to identify approaches for model-checking real-time Java code, which then was investigated more thoroughly in the next phase of the project. In this study, a typical real-time example called the Fischer protocol [16] was implemented in both RTSJ and current real-time model-checkers' languages, to compare the languages and approaches, and also to investigate a way of defining properties in real-time Java programs.

The real-time model-checkers, discussed in Section 2.2, can be grouped into three categories. UPPAAL, Kronos and Rabbit are all based on similar formalisms. They all accept models described in Timed Automata and properties defined in TCTL. There are only a few differences between them. For example, the Timed Automata accepted by Rabbit is a Cottbus Timed Automata which is a subset of UPPAAL's Timed Automata. RT-Spin and MOCHA are both based on languages of distributed modules, so they are

naturally closer to RTSJ. JPF is a Java code model-checker.

Initially we selected one tool from each group to study a range of approaches. UPPAAL was selected as a real-time model-checker from among UPPAAL, Kronos and Rabbit for the following reasons:

- UPPAAL has a graphical user interface;
- UPPAAL has a simulator for the model;
- UPPAAL is well-used and well-supported.

RT-Spin was selected as a real-time model-checker, from among RT-Spin and MOCHA, because properties in MOCHA are described by ATL, whereas RT-Spin has a translator that translates the LTL properties to Promela. There are no time constraints in LTL and timing constraints have to be added by using clock variables which is similar to Java assertions. Moreover, having one language to describe both the system and properties is naturally closer to RTSJ.

In this research, expanding the JPF proposal for model-checking RTSJ for verifying timing properties was not investigated in more detail for the following reasons:

- The approach based on JPF to verify real-time Java code has not been implemented.
- RTSJ features such as ScopedMemoryArea and `NoHeapRealtimeThread` that cannot be the target of the garbage collector, are not supported in the JPF proposal.
- JPF uses a specialized JVM which is called JPF-JVM. JPF-JVM uses backtracking to cover all possible paths by means of symbolic execution (symbolic execution represents values of program variables with

symbolic values instead of concrete data). Therefore, the RTSJ code running under JPF-JVM has to be repeatable and deterministic. As a result, Java's real-time clock is replaced by the simulation clock and it does not have access to a real-time clock. This allows JPF to continue to verify the properties it could verify for Java code (e.g. deadlocks or missed signals, null-pointer references and type-cast errors), in RTSJ. However, with this approach, issues related to JVM behaviour such as process load and thread priorities cannot be checked

3.1 Fischer Protocol

In the Fischer protocol, n processes attempt to access a shared resource; no two processes can access it at the same time. To solve this problem using the Fischer protocol, the system is composed of a set of n timed processes (identical up to renaming), plus a shared variable `id` ranging from 0 to n (the variable `id` is initialised to 0 to indicate the resource is free). Figure 3.1 represents the pseudo-code of process `i` in the Fischer protocol. Each process `i` checks that the common resource is free by checking that `id` is equal to 0 and if so, before `k` time units sets `id` to its own identifier (`i`). Next, it waits for at least `t` time units, checks whether `id` is still equal to `i`, and if so enters the critical section. The Fischer protocol ensures mutual exclusion iff $\texttt{k} \leq \texttt{t}$ [16].

An implementation of the Fischer protocol with RT-Promela is presented on the RT-Spin home-page [37]. This example was used in this study and two other versions with UPPAAL's Timed Automata and RTSJ were implemented as part of this project and are discussed below.

```
Process(i){
         while(true)
         {
             if(id == 0)
             {
                     before k time units id = i;
                     if((id == i) after t time units)
                     {
                         enter critical section;
                         exit critical section;
                         id = 0;
                     }
             }
        }
  }
```

Figure 3.1: Fischer Protocol Pseudo-code

3.1.1 RTSJ Implementation

Figure 3.2 shows the real-time Java code for the process i in the Fischer protocol.

There are three types of clocks in RTSJ which are derived from an abstract class called Clock:

- A *monotonic* clock progresses at a constant rate, suitable for timeouts.
- A *countdown* clock can be reset to zero, paused or continued.
- A *CPU execution time* clock counts the amount of time that is being consumed by a particular thread.

In RTSJ, time is defined by three classes:

- A duration measured by a particular clock is *relative* time.
- *Absolute* time is a time relative to some epoch, such as system start-up time.
- *Rational* time is a subclass of relative time to represent the rate of certain event occurrences.

Each Process `i` initiates a clock variable named `x` and three relative times which are `k`, `t` and `WaitInCs`. `k` and `t` correspond to the variables `k` and `t` in the Fischer Protocol, and `WaitInCs` is the minimum amount of time that process `i` remains in the critical section. An absolute time `at1` is initiated and set to the value of the clock variable, `x`, before the process sets `id` to `pid`. Then, the process checks if the time which passes after setting the `at1` variable is smaller than `k`, which is slightly different from the pseudo-code. Since in RTSJ it is hard to ensure that an action happens before a certain amount of time, in the code first `id` is set to the process's `pid`, and then it is checked that this happens in less than `k` time units. If more than `k` time units have elapsed, it sets `id` back to 0, provided `id` was not set by another process. Note that, if on line 19, `k` is set too small, the process may never enter the critical section. If `id` was set within `k` time units, the process waits for at least `t` time units and ensures that the `id` is still equal to its `pid` before entering the critical section (line 24). Next, the process waits for at least `WaitInCs` time units before leaving the critical section (line 30).

3.1.2 UPPAAL

UPPAAL model checks networks of Timed Automata. Figure 3.3 represents a Timed Automaton process i in the Fischer protocol. The process has four

```
1. public class Process extends Thread{
2.    public static int id;
3.    public static int InCrit;
4.    private int pid;
4.    Clock x;
6.    public Process(int _id){
7.         pid= _id;
8.    }
11.   public void run(){
12.      RelativeTime k = new RelativeTime(0,12000);
13.      RelativeTime t = new RelativeTime(0,10000);
14.      RelativeTime waitInCS = new RelativeTime(4,0);
15.      while(true){
16.               AbsoluteTime at1 = x.getTime();
17.               if(id == 0){
18.                        id = pid;
19.                        if(((x.getTime().subtract(at1)).compareTo(k) >= 0)){
20.                        //   if time is more than k time units undo your action
21.                                  if(id==pid) id = 0;
22.                        }else{
23.                             at1 = x.getTime();
24.                             while(((x.getTime().subtract(at1)).compareTo(t) < 0));
25.                             //   wait for at least t time units
26.                             if((id == pid)){
27.                                       InCrit++;
28.                                       assert(InCrit==1);
29.                                       at1 = x.getTime();
30.                                       while((x.getTime().subtract(at1)).compareTo(waitInCS)<0)){};
31.                                       InCrit--;
32.                                        id = 0;
33.                       }
34.               }
35.     }
36.   }
37.}
```

Figure 3.2: RT-Java code modelling Fischer Protocol

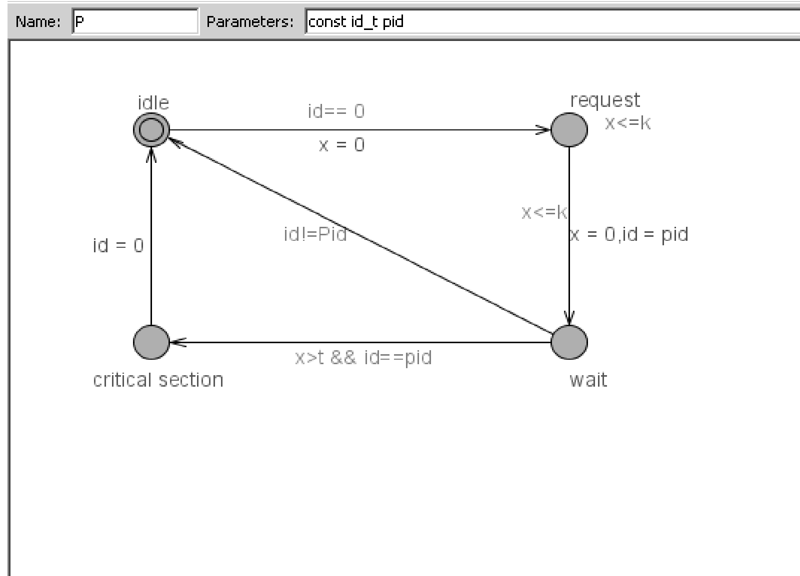

Figure 3.3: Fischer protocol modelled in UPPAAL

locations: *idle*, *request*, *wait* and *criticalsection*. Each process has its own local clock x. However, all clocks progress synchronously.

If the process tries to enter its critical section and id is 0, then the process goes to the *request* state (the transition from *idle* to *request* represents the checking of id, where $id == 0$ is the guard and $x = 0$ sets the clock). The invariant $x <= k$ in the *request* state forces the process to leave the state before k time units. Therefore, the process sets id to its pid within k seconds. If id remains pid for more than t seconds, the process can enter its critical section. However, if id changes before this time, the process returns to the idle state.

UPPAAL's verifier checks for invariants and reachability properties which are expressed in TCTL. It does so by exploring the state space of a system using on-the-fly techniques. For this example, the mutex requirement which checks whether two processes can enter their critical section at the same time

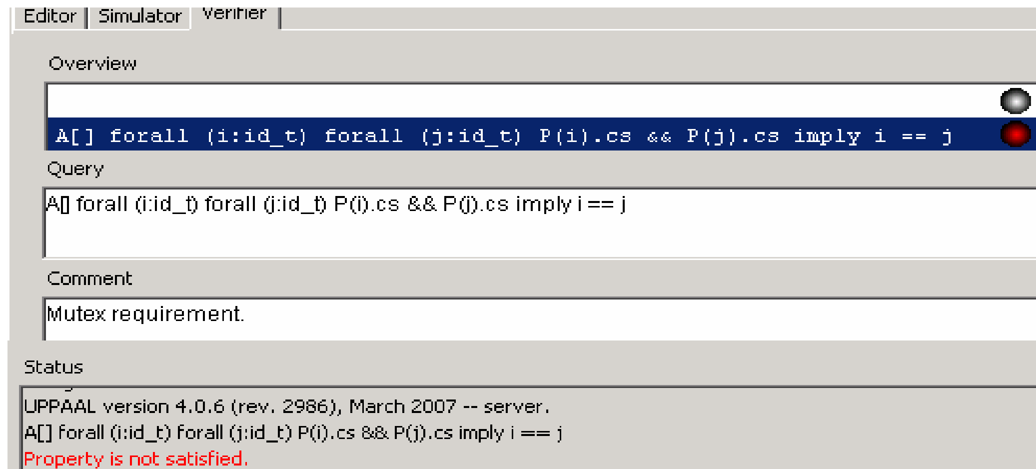

Figure 3.4: Mutex query and the result of its verification

and the result of its verification are shown in Figure 3.4. Figure 3.4 shows that UPPAAL is able to detect a property violation when $k > t$. UPPAAL is also able to show that the property is satisfied when $k \leq t$. In case that a property is not satisfied, UPPAAL provides a counter-example in the form of the shortest trace that violates the property. This trace contains the system Automata and the trace that involves the property. Then, the modeler can use the trace to step through the model from the initial state to the state in which the property is violated.

3.1.3 RT-Spin

RT-Spin model checks RT-Promela code. Figure 3.5 shows the RT-Promela code of process i in the Fischer protocol (this example is presented on the RT-Spin home-page [37]). Clock variables are declared globally in RT-Promela. Therefore, in this example, an array of clocks (`Clock x[N]`) is declared globally. `reset{x[id]} id==0`, expresses that if `id` is 0, process i resets its clock

```
 1. #define N   4       /* number of processes */
 2. #define k  3
 3. #define t  2

 4. clock   x[N];

 5. int id, incrit;

 6. proctype P(byte pid) {
 7.    do ::
 8.      reset{x[id]} id==0 ->
 9.         when{x[id]<k} reset{x[id]} id=pid ->
 10.              when{x[id]>t} id==pid; incrit++; ->
 11.                     id=0; incrit--;
 12.    od
13.}
```

Figure 3.5: Fischer Protocol modelled in RT-Promela

and moves to the next line of the code (line 9). In this part, for a time less than `k` time units, process i sets `id` to its own id (`pid`), resets its clock and moves to the next line. These expressions are all treated as atomic statements, such that as soon as time is checked, `id` is set to `pid`. Line 10 of the code expresses that after at least `t` time units, if `id` still equals `pid`, process i enters the critical section, which is modelled by incrementing `incrit`. Finally, process i leaves the critical section (`id = 0; incrit--`).

The LTL property for the Fischer protocol mutex requirement is (`[]incrit` $\leq$ `1`), which states that the number of processes in the critical section always has to be less than or equal to one (`[]` means always). The Promela code for this requirement, shown in Figure 3.6, is automatically generated by the RT-Spin translator, from the LTL property. The never-claim block goes to the `ErRoR` state iff the number of processes in the critical section becomes more than one (`incrit > 1`).

```
#define ErRoR   assert(0)
never{ skip ->
       do
            :: incrit>1 -> ErRoR
            :: else
       od
}
```

Figure 3.6: Mutex query

The property is satisfied when `k` $\leq$ `t` and RT-Spin also provides a counter-example in a file with a *.trail* extension for the case that `k > t`. This trail file contains every change in the values of the model's variables, from the initial state until the property violation time. The modeler has to trace the value changes to figure out the problem, which is not easy when the trail file is large.

3.2 Conclusion

To increase the confidence in real-time Java code, the formalisms for modelling timing systems and a number of model-checkers were studied. UPPAAL and RT-Spin were selected as the real-time model checkers to investigate more closely. Then, to increase our understanding of RTSJ, and also to become familiar with Java RTS, UPPAAL and RT-Spin, a typical real-time example called the Fischer protocol [16] was implemented, simulated and verified with RTS, RT-Spin and UPPAAL.

In the next phase of this project, a gear controller [17] will form the basis of a more substantial case study. This gear controller is an industrial case

study which requires many interesting RTSJ features for its implementation. An UPPAAL Timed Automata which models this gear controller is also provided by Lindahl et al. [17].

A real-time model is a simplified representation of a real-time system. Models focus on system behaviour and abstract many details of programs [15]. Therefore, in the gear controller case study, we will take a model-based approach to generate RTSJ code from the UPPAAL model controller. In other words, we will start from the existing UPPAAL model, verify it mechanically with UPPAAL and investigate an approach to design RTSJ code which still satisfies the properties.

While the UPPAAL language provides a suitable high-level language for modelling real-time systems, the RT-Promela language is closer to programming languages. There are a couple of tools which translate a subset of Java or Java-like languages to Promela [38, 39]. As an example, JPF version 1 is a translator from Java to Promela [39]. So, as a second part of the gear controller case study we will investigate a mapping from the RTSJ features to RT-Promela and use RT-Spin.

In this way, we will investigate two different approaches for model checking RTSJ code, using current model-checkers:

- Modelling the system with a language supported by a model-checker, verify the model and systematically generate RTSJ code from the model.
- Implementing the system with RTSJ, map the code to a language supported by a model-checker and verify it.

Chapter 4

A Non-Trivial example: A Gear Controller

In this phase of the project, to investigate approaches for increasing the confidence in real-time programs written in RTSJ, we used a nontrivial example: a gear controller [17]. This Gear Controller is an industrial case study which requires many interesting RTSJ feature for its implementation. An UPPAAL Timed Automata which models this gear controller was provided by Lindahl et al.

We initially took a modelling approach to the design of RTSJ programs. First, the UPPAAL model of the gear controller was mechanically verified, to check whether it satisfies particular properties. Then, an RTSJ program was derived from the model by following a systematic approach. This approach can increase the confidence in the correctness of the program.

Next, as discussed in Section 3.2, as a second approach for verifying RTSJ, we map the RTSJ code of the gear controller to RT-Promela and verify it using RT-Spin.

This chapter provides a brief description of the gear controller and illus-

trates the two approaches for verifying RTSJ code.

4.1 Gear Controller

The gear controller is a real-time component that receives gear requests from the driver via a communication network, and implements the actual gear change by actuating the clutch, the engine and the gear-box. The gear-box, the clutch, and the engine are the components in the gear controller environment. These components interact over the communication network.

The gear controller also provides services to its users who are either the driver or a dedicated component implementing a gear-change algorithm through its interface. The gear controller users send their requests to the interface and the interface responds when the service is completed. The interface also keeps track of the gear controller status which can be either changing gear or idling.

The gear controller contains four components: interface, gear-box, clutch and controller. The mapping approaches will be illustrated by means of the Clutch component, which is described in detail here.

The Clutch provides services to open or close the clutch in 100 to 150μs. In the case that opening or closing the clutch takes more than 150μs, the clutch will stop in an error state. Figure 4.1 shows the Timed Automata used by the UPPAAL model checker for a clutch specified by Lindahl et al. [17].

Channels in Timed Automata are used to synchronize and communicate. For `Channel CName`, `CName?` represents receiving a message and `CName!` represents sending a message. In Figure 4.1, a message is sent via `OpenClutch!` and two messages are received via `ClutchIsOpen?` and `ClutchIsClosed?`. `OpenClutch?`, `ClutchIsOpen!` and `ClutchIsClosed!` are in another Timed

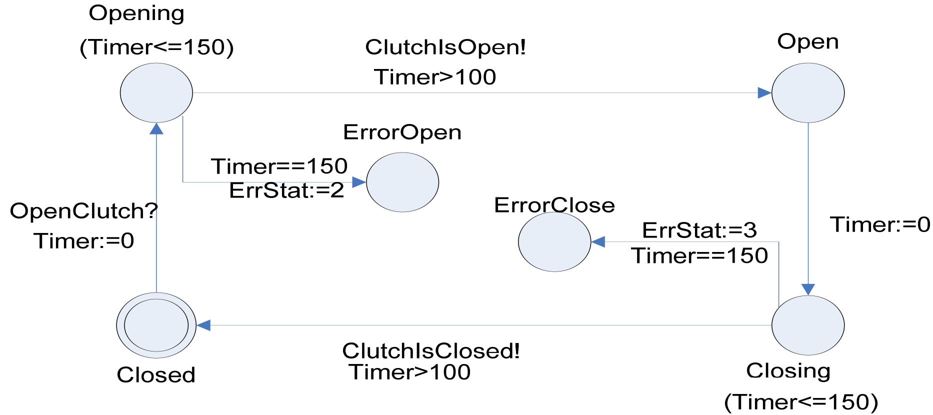

Figure 4.1: Timed Automaton representing Clutch

Automaton not shown in Figure 4.1. `Timer` is a clock and `Timer==150` is a guard. The transition from the `Opening` to `ErrorOpen` can be taken if and only if `Timer==150` is enabled. In other words, the clutch only reaches `ErrorOpen` if `Timer==150`. In the `Opening` state, `Timer<=150` is an invariant. The Automaton needs to leave `Opening` before this invariant is violated.

4.2 Model-based approach

A Model is a simplified representation of the system. We use Timed Automata to describe models. These models represent the behaviour of real-time programs written in RTSJ and they can be verified mechanically with the UPPAAL model-checker. Then, we apply a systematic translation on these models to design real-time programs which still satisfy the properties. Figure 4.2 shows an overview of this model-based approach, which is simi-

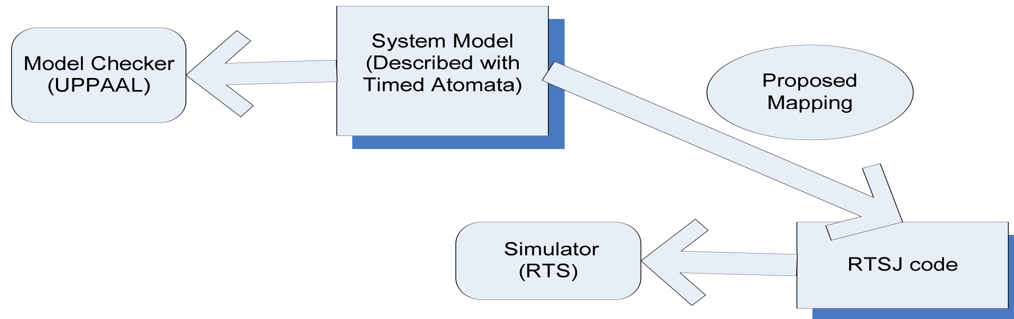

Figure 4.2: Architecture

lar to the model-based approach proposed by Magee and Kramer to design concurrent Java programs from FSP models [15].

4.2.1 Overview

Figure 4.3 shows part of the RTSJ code for the clutch Timed Automaton; the details of this mapping are discussed in Section 4.2.2.

`Clutch` is a non-heap real-time thread. Two classes, `NonHeapRealtimeThread` and `RealtimeThread`, are defined in RTSJ to support real-time threads. Non-heap real-time threads are not targeted by the garbage collector [2].

`ClutchClock` is declared as a clock. Recall from Section 3.1.1, that clocks in RTSJ are derived from an abstract class called Clock. `MaxTimeClutch` and `CurrentTime` are a relative and absolute time respectively (recall that in RTSJ, time is defined by three classes: `relative`, `absolute` and `rational`).

An overview of the mapping for different features and expressions in UPPAAL Timed Automata is shown in Tables 4.1 and 4.2.

```
1.public class Clutch extends NonHeapRealTimeThread{
2.   public void run(){
3.        Environment env = new Environment();
4.        Clock ClutchClock = Clock.getRealtimeClock(); // Timer
5.        RelativeTime MaxTimeClutch = new RelativeTime(0,150);
6.        RelativeTime MinTimeClutch = new RelativeTime(0,100);
7.        AbsoluteTime CurrentTime = ClutchClock.getTime();
8.        String state = "Closed";
9.        while(true){
10.            if(state == "Closed"){
11.                if(env.IsReadyOpenClutch){
12.                   env.IsReadyOpenClutch = false;
13.                   env.ChannelAcknowledgeOpenClutch = true;
14.                   CurrentTime = ClutchClock.getTime();
15.                   state = "Opening";
16.                   continue;
17.                 }
18.             }
19.             if(state == "Opening"){
20.                 if(((ClutchClock.getTime().subtract(CurrentTime)).compareTo(MaxTimeClutch) >= 0)){
21.                    env.ErrStat = 2;
22.                    state = "ErrorOpen";
23.                    continue;
24.                 }
25.                 if(((ClutchClock.getTime().subtract(CurrentTime)).compareTo(MinTimeClutch) > 0)){
26.                    env.ChannelAcknowledgeClutchIsOpen = false;
27.                    env.IsReadyClutchIsOpen = true;
28.                    while(!env.ChannelAcknowledgeClutchIsOpen);//busy loop
29.                    state = "Open";
30.                    continue;
31.                  }
32.             }
33.             if(state == "Open"){...}
34.             if(state == "Closing"){...}
35.        }/*while*/
36.   }/*run*/
37.}/*class*/
```

Figure 4.3: RTSJ code modelling Clutch

Table 4.1: Features Mapping Table

Feature	**Description**	**Currently supported**	**Mapped to**
Timed Automaton	a finite-state machine extended with clock variables	Yes	Real-time Thread
Broadcast channels	channels that are not blocking	Yes	A variable in the `Environment` class
Binary synchronisation	channels are declared as `chan c`	Yes	Two variables in the `Environment` class
Urgent location	time is not allowed to pass in an urgent location	Yes	Resetting the value of the Clock
Urgent synchronisation	delays must not occur if its channel is enabled	No	
Committed location	a state that cannot delay	Partially	Using RTSJ Priorities
Initialisers	used to initialise integers and arrays	Yes	Assignments in the thread constructor

Table 4.2: Expressions Mapping Table

Expression	**Description**	**Currently supported**	**Mapped to**
Assignment	an expression with a side-effect	Yes	An assignment in RTSJ
Guard	a side-effect free expression associated with a transition	Partially	An `if` condition
Invariant	a side-effect free expression associated with a state	Partially	An `if` condition except for time invariants

4.2.2 Mapping details

Timed Automaton: Every Timed Automaton is mapped to a non-heap real-time Java thread. As non-heap real-time Java threads are not targeted by the garbage-collector, programs using such threads have no non-determinism due to garbage-collection delays or memory allocations. Each thread has a state variable that is initialised to the initial state of the Automaton. The behaviour for the Automaton is encoded in an infinite loop in the thread `run` method. This loop contains several `if` statements on the `state` variable and each `if` statement contains the behaviour of the Timed Automaton in a state with at least one outgoing edge. As an example, Figure 3.2 shows the real-time thread corresponding to the Clutch Timed Automaton. Inside the `run` method of the Clutch thread in Figure 3.2, the string variable `state` represents the state, which is initialised to `Closed`. The infinite while loop contains four `if` statements corresponding to the four states with at least one outgoing edge: `Closed`, `Open`, `Closing` and `Opening`.

Global Variables and Broadcast Channels: To model global variables, one additional class, `Environment`, is introduced to implement the environment. The `Environment` contains global variables as static variables and all threads that need to access global variables create an instance of the `Environment` object. Broadcast Channels are considered as global variables, since they are non-blocking. For example, in the `run` method of the Clutch thread in Figure 3.2, an instance of `Environment` is created (line 3) to access shared variables such as `ErrStat`.

Binary synchronisation: In order to model a synchronous channel `C`, two boolean variables are introduced, `IsReadyC` and `ChannelAcknowledgeC`. `IsReadyC` is set to *true* by the sender to inform the receiver that a new message is put in the channel `C` and receiver sets this boolean to *false* whenever

it reads a new value from the channel variable. The `ChannelAcknowledgeC` ensures that the sender will not progress until the receiver receives the message. Whenever the sender sets its channel variable, it also sets the `ChannelAcknowledgeC` to *false* and will not continue until this variable is *true* again. Receiver sets this `ChannelAcknowledgeC` to *true* when it has read the message. The initial value of `ChannelAcknowledgeC` and `IsReadyC` are *true* and *false* respectively. In Figure 3.2, the clutch is the receiver for the `OpenClutch` channel. A transition from `Closed` to `Opening` is taken when a new message is put in the `OpenClutch` channel (line 11). When the clutch receives the `OpenClutch` message, it sets `IsReadyOpenClutch` to *false* to be ready for the next message and also sets `ChannelAcknowledgeOpenClutch` to *true* to inform the sender that it received the message (lines 12 and 13). On the other hand, the clutch is the sender for the `ClutchIsOpen` channel. It sets the `IsReadyClutchIsOpen` and `ChannelAcknowledgeClutchIsOpen` variables (lines 26 and 27) and it waits until the receiver receives this message (line 28).

Urgent Locations: Time is not allowed to pass when the system is in an urgent or committed location. For a Timed Automaton, this is semantically equivalent to a location with incoming edges resetting the Timed Automaton clock and labelled with the invariant `Clock<=0`. However, interleaving with normal states are allowed. To model urgent locations we will add an assignment that saves the value of the clock after all lines of code that lead to an urgent location and then set back the clock to this value when the program leaves the code corresponding to such a location. However, the discrepancy between model and code must be noted and analysed. This feature does not occur in the gear-controller example.

Urgent Synchronisation: In an Urgent Synchronisation, if a synchroni-

sation transition on an urgent channel is enabled, delays must not occur. In RTSJ, a priority scheduler is defined and the priority of an object that extends the Schedulable class can be set. However, even running the object with the highest priority will take some amount of time after it is enabled. The problem is even more challenging when the model contains more than one Urgent Synchronisation. We have not dealt with this feature as it did not occur in the gear-controller example.

Committed Locations: Committed Locations are urgent Locations that can not be delayed when they are enabled. Therefore, the discrepancy between model and code must be noted and analysed. This feature occurred in one Automaton, `Controller`, of our example [17]. The RTSJ code for this Automaton is available online [40].

Clocks: Each instance of a clock in a Timed Automaton is mapped to a clock in RTSJ. To check an upper- or lower-bound on a clock, a relative time is declared in Java for each bound. In addition, every thread contains an absolute time and a clock. To check the time elapsed from a particular moment, the absolute time is set to the current value of the clock. Then, the difference between the current value of the clock and the absolute time will be checked with the corresponding relative time. For example, a clock, two relative times and an absolute time are introduced for timing issues in Figure 3.2 (lines 4-7). In the Clutch Timed Automaton, when the transition from `Closing` to `Opening` is taken, the clock will be reset. The corresponding code for this action is shown on line 14, in which the absolute time `CurrentTime` is set to the current value of the clock `ClutchClock`. Therefore, the time elapsed from this moment will be measured. Inside the `else` statement, the program checks if the time is more than $100\mu s$ (line 25).

Guards: A transition from one state to another state can be taken if and

only if the guard on the transition is enabled. A Guard is translated to an `if` statement. The code inside the `if` block corresponds to the transition updates (assignments). In Timed Automata constraints on the value of the clocks or clock differences are only compared to integer expressions; guards over clocks are essentially conjunctions [41]. Line 20 in Figure 3.2 indicates that if the time elapsed since the last time at which the variable `CurrentTime` is set is equal to or more than the relative time `MaxTimeClutch`, 150, the `ErrStat` will be set to 2 and the `state` will be set to `ErrorOpen`. However, in the Timed Automaton, the guard on the transition from the `Opening` to the `ErrorOpen` is `Timer==150` and not `Timer >= 150`. Since there is no guarantee that the thread will execute this code at exactly the right time, we need to be more flexible in the code than in the model. However, in this case, rather than noting and analysing the difference between model and code, we can actually modify (re-engineer) the model to match the code and then repeat the analysis of the properties we want to check on the modified model.

Dealing with non-determinism: A Timed Automaton can contain more than one transition with an enabled guard. If a state contains more than one outgoing edge with an enabled guard, one of them will be taken non-deterministically. Following the standard notion of refinement, an implementation can be more deterministic than the model. However, if we want to implement the non-determinism we can use a random variable in RTSJ. This feature occurred in one Automaton, `Interface`, of our example [17]. The RTSJ code for this Automaton is available online [40].

Invariants: The Automaton needs to leave a state before its state invariant is violated. In other words, Timed Automaton must take one of the enabled transitions if the current state invariant is violated. In RTSJ we

cannot guarantee that the thread has a CPU before a certain time limit. To deal with this in RTSJ code we add assumptions on upper-bound of the run time of RTSJ code. We can accumulate the upper-bound of the run time of RTSJ code. To accumulate this run time upper-bound we can assign a fixed RTSJ run-time (based on the version of RTSJ and the hardware we use) to each line of code. We can then add these times to accumulate the run time of code corresponding to a state with an invariant. In Figure 4.1, the `Opening` has a state invariant, `Timer<=150`. Therefore, the clutch cannot stay in the `Opening` state for more than 150 ms and it needs to go to either `ErrOpen` or `Open` before this invariant is violated. This invariant is an assumption that should independently verified for a particular hardware and version of the RTSJ to check opening the clutch should take no more than 150 ms. In other words, executing the code on lines 14-16, 19-20 and 25, and also the code on lines 14-16 and 19-21 in Figure 3.2 should take less than $150\mu s$.

4.2.3 Verification result

To illustrate the applicability of the proposed method, we applied this approach to the gear-controller [17]. The model presented by Lindahl et al. contains 5 Timed Automata with a total of 63 states and 83 transitions. We recreated this model and verified it with UPPAAL. However, the verification results were not entirely consistent with the result provided by Lindahl et al. [17]. We had to add the timing invariants on all states and increase the time bounds in the timing properties to satisfy them.

The RTSJ derived from the Timed Automaton had 5 Java threads, 1320 lines of code and 16 assumptions for 16 time invariants in the model. The Timed Automata for the gear controller and the RTSJ code are both available

online [40].

We unintentionally made an error in the UPPAAL model (when the clutch Automaton transitions from `Opening` to `ErrOpen`, we did not set `ErrStat` to 2). As a result, one of the system properties was violated. This property required the gear controller to notice that the clutch reached `ErrOpen`, before $300\mu s$. UPPAAL detected this error and we fixed the model. We wanted to see whether the same error would be detected in RTSJ. Therefore, we removed line 22 from Figure 3.2. However, the error was not detected since the offending code was not executed in the simulator as the timer never exceeds $150\mu s$. Then, we changed the invariant on `Opening` from $150\mu s$ to $50\mu s$ (line 20) and the error was detected. This shows why the model-checking approach is useful, as it detected an error in the model that is more difficult to detect in the code.

To demonstrate the discrepancy between the model, in which we make assumptions about invariants, and the code, where lines of code take a certain amount of time to execute, we changed the timing in both the model and implementation. In the original model, the invariant on the `Opening` state is $150\mu s$ and the transaction to `Open` can only be made after $100\mu s$. These properties are used to prove that if the clutch transitions to `ErrOpen`, then the gear controller notices this before $300\mu s$. We then changed the invariant from $150\mu s$ to $2\mu s$ and the guard on the transition to `Open` to $1\mu s$. In the model this is still sufficient to prove the gear controller notices the error before $4\mu s$. However, if we make these changes in the code, then the error is only detected after $50\mu s$.

This approach has some limitations. As an example, when the model contains specific time values, rather than upper- or lower-bounds, it cannot be straightforwardly mapped to RTSJ code. Some other features, such as

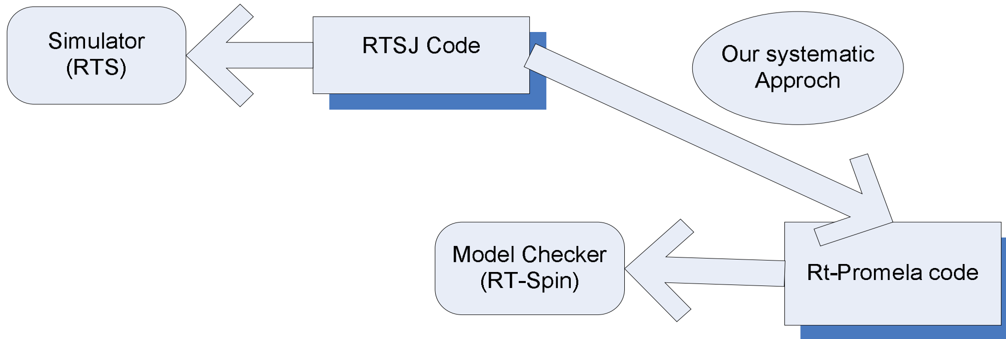

Figure 4.4: Architecture

urgent synchronisation, were also left for future work.

4.3 From RTSJ to RT-Promela

This section discusses an approach to generate RT-Promela code from RTSJ code. These RT-Promela programs can be verified mechanically with the RT-Spin model-checker. Figure 4.4 shows an overview of this approach, which is similar to the approach proposed by Havelund to translate Java code to Promela in JPF version 1 [39].

4.3.1 Overview

To avoid the complexity of the translation of all of RTSJ to RT-Promela, we used an automated tool, R2P [42], which translates a Java-like language, Rebeca [42], to Promela. We first removed the RTSJ features from the RTSJ code and generated the Promela code, using R2P. We then investigated the mapping between RTSJ features that we used in the gear controller code to RT-Promela and added them to the generated Promela code. Figure 4.5 shows the RT-Promela code generated from the RTSJ code shown in Fig-

ure 4.3. In this figure the lines starting with '*' were added manually while the rest of the code was derived by using R2P.

In this case study, we only considered those RTSJ features that occurred in our derived RTSJ code from the UPPAAL model. The details of the mapping are provided below.

4.3.2 Mapping details

Real-Time Threads: Recall that there are two RTSJ type of threads, but only `NoHeaptimeThread` occurred in the code. Each non-heap real-time thread in RTSJ is translated to a `Proctype` in RT-Promela. Figure 4.5 shows the `Proctype reac_Clutch` corresponding to the Clutch thread in Figure 4.3. All the variables in RT-Promela are declared globally. As an example, `s_Clutch_state` corresponds to the variable `state` in the RTSJ code. The `run` method of the Clutch Proctype in Figure 4.5 contains four `if` statements corresponding to the four if statements in the `run` method of the RTSJ code (lines 2-14 in Figure 4.5 correspond to lines 10-18 in Figure 4.3).

Relative and Absolute Times: Recall that there are three RTSJ time classes, but them only `Relativetime` and `Absolouttime` occurred in the code. All RTSJ `relative` time variables are mapped to integer values in RT-Promela, which are used for initiating and/or comparing with clock values. As an example, the value 150 in line 18 of Figure 4.5, represents `MaxTimeClutch` in RTSJ and corresponds to line 20 in Figure 4.3.

Mapping `absolute` time is more challenging. Recall that `absolute` time is a time relative to some epoch, such as the time at which the execution is started. In RT-Promela there is no such concept. As mentioned in Section 4.2.2, we used this `absolute` time, `CurrentTime`, to check the time elapsed since a particular moment (instead of resetting the clock). Therefore, we

```
proctype reac_Clutch(byte p_myIndex; byte p_myID;
 1.:: reac_ClutchChan[p_myIndex] ? [reac_Clutch_ms_run, sender] -> /* serving run */
 2.      if
 3.      :: s_Clutch_state[p_myIndex] == 1 ->
 4.          if
 5.          :: s_Clutch_OpenClutch[p_myIndex] ->
 *6.              reset{CLOCK[0]} s_IsReady_OpenClutch[p_myIndex] = false;
 7.                             s_ChannelAcknowledge_OpenClutch[p_myIndex] = true;
 8.             s_Clutch_state[p_myIndex] = 2;
 9.          :: else ->
 10.            skip;
 11.        fi;
 12.    :: else ->
 13.        skip;
 14.    fi;
 15.    if
 16.    :: s_Clutch_state[p_myIndex] == 2 ->
 17.        if
 *18.            :: when{CLOCK[0] >= 150 when{CLOCK[0] <= 150}}->
 19.              s_Clutch_ERRSTAT[p_myIndex] = 2;
 20.              s_Clutch_state[p_myIndex] = 4;
 *21.            :: when{CLOCK[0] > 100 when{CLOCK[0] <= 150}}->
 22.              s_IsReady_OpenClutch[p_myIndex] = true;
 23.              s_ChannelAcknowledge_OpenClutch[p_myIndex] = false;
 24.              if
 25.               W1: :: s_ChannelAcknowledge_OpenClutch[p_myIndex] == false-> goto W1;
 26.                   :: else->
 27.                        s_Clutch_state[p_myIndex] = 3;
 28.              fi;
 29.        fi;
 30.      :: else ->
 31.          skip;
 32.      fi;
 33.      if
 34.      :: s_Clutch_state[p_myIndex] == 3 ->...
 35.      fi;
 36.      if
 37.      :: s_Clutch_state[p_myIndex] == 6 ->...
 38.      fi;
 39.      reac_ClutchChan[p_myIndex] ! reac_Clutch_ms_run, p_myID;
 40.      reac_ClutchChan[p_myIndex] ? _, _
```

Figure 4.5: Promela code modelling Clutch

reset the clock in RT-Promela when `CurrentTime` is set to the current value of `ClutchClock` (the corresponding code for line 14 in Figure 4.3 is line 6 in Figure 3.5 in which `Clock[0]` is reset). Although this is not a universal and automatic solution for mapping `absolute` time to RT-Promela, this worked for the code we derived from the model. However, a more general solution would be needed for mapping `absolute` time.

Clocks: Recall from Section 3.1.1, that clocks in RTSJ are derived from an abstract class called `Clock`. Each instance of a clock in RTSJ is mapped to a clock in RT-Promela.

Assumptions: Recall that we dealt with the time invariants by adding assumptions. As an example, for a particular hardware and version of the RTSJ, opening the clutch should take no more than 150 ms (executing the code on lines 14-16, 19-20 and 25, and also the code on lines 14-16 and 19-21 in Figure 4.3 should take less than $150\mu s$). For each of these assumptions, we need to put a proper upper-bound time limit in the RT-Promela code. Otherwise, the model can stay in one state forever and timing properties will be violated.

For this particular example, as we had enough knowledge about the invariants in the original UPPAAL model, adding these upper-bounds was straightforward (e.g. `when{CLOCK[0] <= 150}` on lines 18 and 21). However, a more general solution would be needed to deal with such assumptions.

4.3.3 Verification result

The RT-Promela code generated from the RTSJ code is available online [40]. There are 1014 lines of code, of which 916 lines were generated by R2P.

We ran RT-Spin on the generated RT-Promela. All the properties were

satisfied. We then used our approach to derive RT-Promela code from the faulty version of the code presented in Section 4.2.3, to investigate whether RT-Spin can capture the error that was captured by UPPAAL. Therefore, we removed line 25 from the RT-Promela code. As expected, RT-Spin detected the error.

As in Section 4.2.3, we decreased the timing bounds in the RT-Promela code to show the discrepancy between the model and code due to our assumptions. Recall from Section 4.2.3, in the original model, the clutch opens in 100 to 150μs. In the case that opening the clutch takes more than 150μs, the clutch should stop in an error state. One of the properties required the gear controller to notice the clutch reached `ErrOpen`, before 300μs. As for the RTSJ code, we decreased the 150μs to 2μs and the 100μs to 1μs. This was enough to verify the property that the gear controller notices the error before 4μs. However, as mentioned previously, in the RTSJ code, the time difference between the time the error happened and the time at which gear controller noticed it, was around 60μs.

4.4 Conclusion

Based on our experience with the case study, the model-based appears be the most promising approach. This approach is based on UPPAAL models and mapping to RTSJ code and a significant portion of UPPAAL models can be translated with potential for partial automation.

On the other hand, mapping RTSJ `absolute` time and assumptions to RT-Promela are challenging. More importantly, in this case study, we only mapped the code which was derived from the UPPAAL Timed Automata and

did not cover any other RTSJ features. Generating RT-Promela code from arbitrary RTSJ code will be much more challenging. As an example, mapping a feature such as `scoped Memory`, which is a subclass of `Physical Memory` and can be subject to garbage collection delays, would be very difficult.

Therefore, we will continue in this project with the model-based approach based on UPPAAL and RTSJ.

Chapter 5

Remainder of the project

The goal of this project is to increase our confidence that RTSJ code satisfies certain properties. The main contribution of this work will be a model-based approach for implementing real-time systems. First, models can be mechanically verified, to check whether they satisfy particular properties, by using UPPAAL. Then, programs are derived from the model by applying the mapping from UPPAAL to RTSJ.

In this research, first, we studied formalisms for modelling timing systems and a number of model-checkers. Then two methods for model-checking RTSJ programs were investigated, which were using a model-based approach to derive RTSJ code from Timed Automata supported by UPPAAL and using a mapping algorithm to map RTSJ code to RT-Promela supported by RT-Spin. Based on the results of this phase of the project, we decided to pursue the model-based approach using UPPAAL and RTSJ in the remainder of the research.

In the next phase of the project, the mapping details from UPPAAL Timed Automata to RTSJ features will be finalised. This mapping is expected to be partial. As discussed in Section 4.3.2, mapping `committed`

`location` and `urgent synchronisation` does not seem be feasible. Therefore, based on the result provided in Tables 4.1 and 4.2, the Timed Automata features for which the translation will be proposed are `broadcast channels`, `binary synchronisation`, `initialisers`, `assignment`, `guard`, `urgent locations` and `invariant`. We will semi-formally prove our mapping. This involves comparing the semantics of UPPAAL Timed Automata with the semantics of RTSJ. Combinational equivalence checking(CEC) [43] and sequential equivalence checking(SEC) [44] are the sort of methods we will explore in this phase. With these methods, the equivalence of models can be proved by determining the correct relation between their state encodings rather than by calculating the entire reachable state-space of the system.

Finally a prototype tool will be developed to translate our subset of the UPPAAL Timed Automata to RTSJ. This tool will accept XML files corresponding to UPPAAL Timed Automata which are generated by UPPAAL and generate the RTSJ code for the model. The transformation will be partially automated by using this code generator. We expect to automatically generate the structure of the implementation. As an indication of the automation, we expect to be able to generate around ninety percent of the Fischer protocol code and sixty percent of the gear controller.

Next, the types of properties which are needed to be checked in the code will be determined, and they will be divided into three categories:

- Properties that are guaranteed to hold unconditionally in the code when they hold in the model.
- Properties that are guaranteed to hold provided that certain conditions are satisfied by the RTSJ code.
- Properties that cannot be guaranteed to hold based on the mapping.

As an example of conditional properties, we need assumptions on the upper-bound of the execution time of the RTSJ code, for checking timing properties, in case the corresponding model contains timing invariants. To accumulate this run time upper-bound, we can assign a fixed RTSJ run-time (based on the version of RTSJ and the hardware we use) to each line of code. We can then add these times to accumulate the run time of the code corresponding to a state with an invariant. For example, as discussed in Section 4.2.2, we had the assumption that with a particular hardware and certain version of the RTSJ, the opening clutch should take no more than 150 ms.

There will also be a subset of properties that cannot be guaranteed to hold based on the mapping. As an example, when the model or property contains timing constraint on specific timing values (rather than upper- or lower-bounds), as there is no guarantee that the thread will execute the corresponding code at exactly one time, we can not guarantee that the timing properties that hold in the model still hold in the derived code. We need to be more flexible in the code than in the model. Recall from Section 4.2.2 that for specific time values in our example, we re-engineered the model to match the code and then repeated the analysis of the properties we wanted to check on the modified model. Alternative V&V techniques can also be used to check these sorts of properties. In the final phase of this research, we will investigate other V&V techniques for verifying the properties that cannot be guaranteed to hold. The sort of V&V techniques that will be investigated are the method is proposed by Blom et al. [45] and adding assertions to the RTSJ code.

5.1 Timetable

The planned timetable for this project is shown in Figure 5.1. In the remainder of the project, first, the mapping approach will be finalised. This phase consists of documenting the mapping, proving the mapping semi-formally, dividing the properties, specifying the conditions for properties that are guaranteed to hold conditionally, prototyping and developing the code generator and finally showing the applicability of our approach and the tool by means of a non-trivial case study. The final case study will preferably be an industrial case study that covers all interesting features of our subset of Timed Automata. The UPPAAL Timed-Automata of the system will be generated. Properties which have to be satisfied will also be identified. This case study will then be translated to RTSJ, using our code generator and following our approach. The defined properties will be checked on both the model and the code.

In the final phase of the project, we will briefly investigate other V&V techniques to strengthen our method. We will use the same case study as in the previous phase to validate approaches.

			2008	2009	2009	2009	2009	2010
Task Name	Duration	Start	1-Oct to 1-Dec	1-Jan to 1-Mar	1-Mar to 1-Jun	1-Jun to 1-Oct	1-Oct to 1-Dec	1-Jan to 1-Mar
Finalizing the mapping	12m	01-Oct-08						
Documenting the mapping	3m	01-Oct-08						
proving the mapping	3m	01-Oct-08						
Dividing properties and specifying the conditions	3m	01-Jan-09						
Tool generation	3m	01-Mar-09						
Case Study	3m	01-Jul-09						
Investigating alternative V&V techniques	3m	01-Oct-09						
Thesis writing	18m	30-Sep-08						

Figure 5.1: Research Timetable

Bibliography

[1] Real-time Java, part 1: Using the Java language for real-time systems. http://www-128.ibm.com/developerworks/java/library/j-rtj1/index.html. Date accessed: 14 June 2007.

[2] Andy Wellings. *Concurrent and Real-Time Programming in Java*. John Wiley and Sons Ltd, 2004.

[3] Jr. Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. MIT Press, 1999.

[4] Thomas A. Henzinger, Xavier Nicollin, Joseph Sifakis, and Sergio Yovine. Symbolic model checking for real-time systems. *Inf. Comput.*, 111(2):193–244, 1994.

[5] P. Bellini, R. Mattolini, and P. Nesi. Temporal logics for real-time system specification. *ACM Comput. Surv.*, 32(1):12–42, 2000.

[6] Johan Bengtsson and Wang Yi. Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets. W. Reisig and G. Rozenberg (eds.), LNCS 3098*, pages 89–90. Springer-Verlag, 2004.

[7] Sergio Yovine. Kronos: a verification tool for real-time systems. *Journal on Software Tools for Technology Transfer*, 1(1–2):123–133, 1997.

[8] Kim G. Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, 1997.

[9] Stavros Tripakis and Costas Courcoubetis. Extending Promela and Spin for real time. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 329–348. Kluwer Academic Publishers, 1996.

[10] Dirk Beyer. Rabbit: Verification of real-time systems. In *Proceedings of the Workshop on Real-Time Tools*, pages 13–21. IEEE Computer Society, 2001.

[11] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani, and Serdar Tasiran. MOCHA: Modularity in model checking. In *Computer Aided Verification*, pages 521–525. Kluwer Academic Publishers, 1998.

[12] Java SE real-time system - evaluation downloads. http://java.sun.com/javase/technologies/realtime/rts/. Date accessed: 20 August 2007.

[13] G. Brat, K. Havelund, S. Park, and W. Visser. Java PathFinder - a second generation of a Java model checker. In *Proceedings of the Workshop on Advances in Verification*, pages 164–169. World Scientific Publishing Company, 2000.

[14] Gary Lindstrom, Peter Mehlitz, and Willem Visser. Model checking real time Java using Java PathFinder. In *3rd Int'l Symposium on Automated Technology for Verification and Analysis*, pages 444–456. Springer-Verlag, 2005.

[15] Jeff Magee and Jeff Kramer. *Concurrency State Models and Java Programs*. John Wiley and Sons Ltd, 2005.

[16] Leslie Lamport. A fast mutual exclusion algorithm. *ACM Trans. Comput. Syst.*, 5(1):1–11, 1987.

[17] Magnus Lindahl, Paul Pettersson, and Wang Yi. Formal design and analysis of a gear controller: an industrial case study using UPPAAL. Technical Report ASTEC 97/09, Advanced Software Technology, Uppsala University, 1997.

[18] Niusha Hakimipour, Paul Strooper, and Roger Duke. Exploring model-based development for verification of real-time Java. In *5th International verification Workshop VERIFY08*, pages 71–81. Springer-Verlag, 2008.

[19] C. Ramchandani. Analysis of asynchronous concurrent systems by Timed Petri Nets. Technical Report TR120, MIT (Massachusetts Institute of Technology), 1974.

[20] Bernard Berthomieu and Michel Diaz. Modeling and verification of time dependent systems using Time Petri Nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273, 1991.

[21] Anton Wijs. Achieving discrete relative timing with untimed process algebra. In *ICECCS '07: Proceedings of the 12th IEEE International Conference on Engineering Complex Computer Systems (ICECCS 2007)*, pages 35–46. IEEE Computer Society, 2007.

[22] Rajeev Alur and Thomas A. Henzinger. A really temporal logic. In *IEEE Symposium on Foundations of Computer Science*, pages 164–169. IEEE Computer Society, 1989.

[23] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[24] P. M. Merlin and D. J. Farber. Recoverability of communication protocols: Implications of a theoretical study. *IEEE Trans. Comm.*, 24(9):1036–1043, 1976.

[25] J. Baeten. Timed Process Algebras. *Electronic Notes in Theoretical Computer Science*, 52(3):1–2, 2004.

[26] Jan F. Groote. The syntax and semantics of timed μCRL. Technical Report R9709, CWI (Centre for Mathematics and Computer Science), 1997.

[27] Stefan Blom, Natalia Ioustinova, and Natalia Sidorova. Timed verification with μCRL. In *Perspectives of System Informatics*, pages 178–192. Springer-Verlag, 2004.

[28] Johan van Benthem. *Modal Logic and Classical Logic.* Bibliopolis, Napoli, 1985.

[29] Patrick Blackburn and Johan Van Benthem. *Modal logic: a Semantic Perspective.* Elsevier, 2006.

[30] M. Gabbay. *Temporal Logic: Mathematical Foundations and Computational Aspects.* Clarendon Press, Oxford, 1994.

[31] Edmund M. Clarke and Orna Grumberg. The model checking problem for concurrent systems with many similar processes. In *Temporal Logic in Specification*, pages 188–201. Springer-Verlag, 1987.

[32] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. In *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pages 414–425. IEEE Computer Society, 1990.

[33] Gerd Behrmann, Kim G. Larsen, and Jacob I. Rasmussen. Optimal scheduling using priced timed automata. *SIGMETRICS Perform. Eval. Rev.*, 32(4):34–40, 2005.

[34] Promela language reference. http://www.cse.msu.edu/. Date accessed: 24 July 2007.

[35] K. Havelund and G. Rosu. Testing linear temporal logic formulae on finite execution traces. Technical Report 59, RIACS (Research Institute for Advanced Computer Science), 2001.

[36] Rajeev Alur, Thomas A. Henzinger, and Orna Kupferman. Alternating-time temporal logic. *J. ACM*, 49(5):672–713, 2002.

[37] Real-time SPIN. http://www-verimag.imag.fr/ tripakis/rtspin.html. Date accessed: 27 June 2007.

[38] Marjan Sirjani, Ali Movaghar, Amin Shali, and Frank S. de Boer. Model checking, automated abstraction, and compositional verification of Rebeca models. *Universal Computer Science*, pages 1054–1082, 2005.

[39] Klaus Havelund. Java pathfinder, a translator from java to promela. In *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*, page 152, London, UK, 1999. Springer-Verlag.

[40] Niusha hakimipour's home page. http://itee.uq.edu.au/ñiusha/GearControler.rar. Date accessed: 1 May 2008.

[41] Gerd Behrmann, Alexandre David, and Kim G. Larsen. A tutorial on Uppaal. www.it.uu.se/research/group/darts/papers/texts/new-tutorial.pdf, 2004. Date accessed: 15 May 2007.

[42] Rebeca homepage. Date accessed: 20.December.2007.

[43] J. Marques-Silva and T. Glass. Design, automation and test in europe conference and exhibition. In *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 145–149. IEEE Computer Society, 1999.

[44] C.A.J. van Eijk. Sequential equivalence checking based on structural similarities. In *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, pages 814–819. IEEE Computer Society, 2000.

[45] Stefan Blom, Thomas Deiβ, Natalia Ioustinova, Ari Kontio, Jaco van de Pol, Axel Rennoch, and Natalia Sidorova. Simulated time for host-based testing with TTCN-3. *Softw. Test. Verif. Reliab*, 18(1):29–49, 2008.