Verification of Real-time Systems

Hugh Anderson and P.S. Thiagarajan

January 5, 2007

Contents

Preface

The following is a specification of a postgraduate real-time systems verification course:

> *This course will provide an introduction to the analysis and verification of hard real time systems. These are systems, typically running embedded distributed applications, that must meet their temporal constraints in a range of anticipated load and fault scenarios. The focus will be on the tools and techniques based on timed automata with which one can verify that the scheduled behavior of a real time distributed system will meet its critical timing constraints. The overall goal is to provide the student with the current scientific and engineering insights that are relevant for the analysis and verification of distributed embedded real time systems.*
>
> *Some of the topics to be covered in the course are: the application domains; hard and soft timing constraints; the system components; distributed real time embedded systems, timed Automata and temporal logics, the UPPAAL verification tool, efficient data structures for state space representation; time-triggered architectures, protocols and their applications.*

The content of this book mostly derives from material presented during the postgraduate course “CS5270 Verification of Real-time Systems” developed by Professor P.S. Thiagarajan, and delivered at the Computer Science Department of the School of Computing at the National University of Singapore.

About the Authors

Hugh Anderson is a lecturer at NUS. He held a similar position at the University of the South Pacific in Suva, Fiji, until the coup in 2000 led him to relocate to Singapore. Hugh has an open-door policy, and encourages his students to visit him at any time. Hugh's prior work experience includes fifteen years of lecturing, and nearly fifteen years of consultancy, during which he developed software and hardware for embedded systems such as petrol pumps, cash registers and so on. In addition, he developed network and administration software for a large X.25 network, and many other jobs he would prefer to forget. Hugh's research interests are in the formal derivation and verification of complex systems.

P.S. Thiagarajan received his Ph.D. in Computer Science from Rice university, Houston, Texas, USA (1973), and has had a career in research at MIT and the Gesellschaft fuer Mathematik und Datenverarbeitung, St. Augustin, Germany, as well as an academic career at the Computer Science Department of Aarhus University, and at the Chennai Mathematical Institute (1989-). He was a visiting professor at Georgia Tech during 2000-2001 before coming to the Computer Science Department of the School of Computing at the National University of Singapore (NUS) where he is currently a Professor. He has been an invited speaker at a variety of international conferences, has lectured in many advanced courses and has held visiting professorships around the world. He is a member of the editorial boards of the journals Theoretical Computer Science and the International Journal on Foundations of Computing. He served two terms (1997 - 2003) as a member of the Governing Council of the European Association for Theoretical Computer Science (EATCS). He is a Fellow of the Indian Academy of Sciences and the Indian National Academy of Sciences.

Chapter 1

Introduction

Aristotle maintained that women have fewer teeth than men; although he was twice married, it never occurred to him to verify this statement by examining his wives' mouths. [Bertrand Russell]

1.1-1.8 Orientation - a range of topics Page 2
We introduce key topics in the verification of real-time systems.

Concepts introduced: Motivation, real-time systems, assurance.

COMPUTER SYSTEMS are found in every facet of modern society. In a typical modern living room, it is possible to find a large number of computer systems performing various tasks. Even a humble remote (infra-red) controller may have an embedded microprocessor in it. Each of these computer systems is running software, and in some cases the software performs a fairly critical operation (perhaps not in a living room). Consider the software inside the computers that control the Singapore MRT (Mass Rapid Transit) North-South train line: an error in this software could result in a dreadful accident if it failed to stop the train at the end of the line. However, we do not need to look for *hypothetical* examples of failure of software. There have been many examples of computer systems that have not operated as expected, resulting in catastrophic failures.

Figure 1.1: The explosion of the Ariane-5

The Ariane 5 launcher blew up in 1996 because of a software error. The error which ultimately led to its destruction about 40 seconds after lift off on its maiden flight was clearly identified in the report of the investigating committee.

The attitude of the launcher and its movements in space are measured by an Inertial Reference System (SRI). It has its own internal computer, in which angles and velocities are calculated on the basis of information from a "strapdown" inertial platform, with laser gyros and accelerometers...

The launcher started to disintegrate at about H0 + 39 seconds because of high aerodynamic loads due to an angle of attack of more than 20 degrees that led to separation of the boosters from the main stage, in turn triggering the self destruct system of the launcher. [Ari]

The underlying cause for all this was a section of Ariane-5 software that was re-used from earlier Ariane-4 launchers. This software was never tested during software development, as it was considered too expensive, and needed real-time input from sensors moving in space. The fault was rooted in time delays, and the inability of the runtime system to handle a software fault. It is likely that if the constraints of the system had been specified, and the software tested against these constraints, then the problem would have been found without an explosion. Instead, a run time error arose in both the active and the backup computers was detected and both computers then shut themselves down. This resulted in the total loss of attitude control. The Ariane 5 turned uncontrollably and aerodynamic forces broke the vehicle apart.

1.1 Motivation

Compare the following two warranties:

> *PC Manufacturer warrants that*
>
> > *(a) the SOFTWARE will* ***perform substantially in accordance*** *with the accompanying written materials for a period of ninety (90) days from the date of receipt, and*
> >
> > *(b) any Microsoft hardware accompanying the SOFTWARE will be* ***free from defects*** *in materials and workmanship under normal use and service for a period of one (1) year from the date of receipt.*

> *ACCTON warrants to the original owner that the product delivered in this package will be* ***free from defects*** *in material and workmanship for the lifetime of the product.*

The first warranty applies to a large *software* product (Windows NT), the product of hundreds of person-years of software development effort, and containing millions of separate components. The second warranty applies to a large *hardware* product (An intelligent hub), the product of tens of person-years of software and hardware development effort, and containing millions of separate components.

Taken together, it is clear that the reliability of software is viewed as poor, even by the largest software companies in the world, in comparison with hardware. One explanation often given for this sad state of affairs is that the software is more complicated than the hardware, and hence there is more opportunity for failure. However, this explanation is not particularly helpful, as it is clear that even small[1] software is inherently unreliable. We have to recognize that most commercial software in use today contains large numbers of errors, and the software industry must improve. Another way of restating this is say that *software engineers* must improve.

Consider the following problem:

> The sum of the ages of my three sons is five times the age of the smallest son. The sum of the ages of the two youngest sons is six years younger than the sum of the ages of the two oldest sons. The oldest son has the next birthday, and when he has it, he will be two years older than twice the age of the youngest son. ***What are the ages of my three sons?***

[1]A *small* software structure in this context might be one with (say) up to 2,000 lines of code, 200 variables and four programming interfaces.

If a school child was presented with this problem, he or she would immediately reach for a piece of paper, and represent the problem in an abstract manner[2]:

$$
\begin{aligned}
a + b + c &= 5 * a \\
a + b &= (b + c) - 6 \\
(2 * a) + 2 &= c + 1
\end{aligned}
$$

Having done this, the set of equations are then reduced using well known methods. The general strategy taught at school is that "*if a problem is too complicated to understand all at once, rewrite it in an abstract notation and then use a set of rules to solve it*".

For contrast, consider the following problem:

A file consists of two sorts of records - unprocessed records and processed records. It contains at least one record, and we cannot tell the type of a record without first accessing it. A program randomly accesses two of the records.

- ❖ If they are of different types, the program amalgamates the data in the records, deletes them, and writes a new unprocessed record back to the file.
- ❖ If the records are of the same type, the program processes the data in the records, deletes them, and writes a new processed record back to the file.

This process continues until the program cannot get two records from the file. Given a starting state of the file: ***Does the program terminate? What is in the file when it terminates?***

If a software engineer is presented with this problem, he or she might "*model it using a small C program*", "*Run it a few times and see what happens*", or perhaps "*Start with a file with one record of each type, then try a bigger file until a pattern emerges*". (Actual responses!) In each case, the software engineer is using a poor strategy. The problem can be easily abstracted and solved.

When chemists, physicists or biologists meet a problem that is too large or difficult to understand, they turn to mathematics for help. When software engineers meet a problem that is too large or difficult to understand, they try to code it a few different ways to see what happens, or perhaps develop a few subassemblies, or ...

[2] Actually, only my oldest son would do this...

1.1.1 Errors or bugs?

The software industry often refers to software *errors* as *bugs*. This trivializes the issue, and gives the software developer a false sense of security (*Once I've removed these last few bugs, it will be perfect!*). In reality, the software developer may stand on very shaky ground: *Once I've removed these last few bugs...*

- *it has been **known to work** for 11 minutes continuously... (!)*
- *it has **no written documentation** describing it's architecture...*
- ***no-one** else **could repair** it without spending months looking at my code...*

The Internet newsgroup **comp.risks** reports on *"Risks to the Public in the Use of Computer Systems and Related Technology"*, and makes interesting reading:

- The Arizona lottery never picked the number 9 (a bad random number generator)
- The LA counties pension fund lost US$1,200,000,000 through programming error.
- A Mercedes 500SE with graceful no-skid brake computers left 200m skid marks. A passenger was killed.
- A woman killed her daughter, and attempted to kill herself and her son, after a computer error led to her being told that all her family had an incurable cancer.
- A computer controlled elevator door in Ottawa killed two people.
- An automated toilet seat in Paris killed a child.
- The Amsterdam air-freight computer software crashed, killing giraffes.
- The Panasonic children's Internet watchdog software regularly emits vulgarities.
- General Motors recalled over 1,000,000 cars for a software change to stop erratic air bag deployment.

There are thousands more of these sort of stories documented over the last 20 or so years, and hopefully somewhere in here you have seen the need for formal systems for verifying complex systems.

1.2 Real-time systems

In this book we focus on the analysis and development of real-time computer systems, so we must begin by defining what we mean by a *real-time* system.

Definition 1 *A real time system is a computer system in which correct functioning depends on both the values of the results produced, and the physical times at which the results are produced.*

A real-time computer system is considered to be embedded in a larger physical system, whose *state* evolves with time. What is meant by state? Well, we may be interested in properties such as position, velocity, and acceleration, or perhaps others such as pressure, temperature, level, concentration of chemicals, and so on. For each of these the state may need to be sensed and controlled by the computer system.

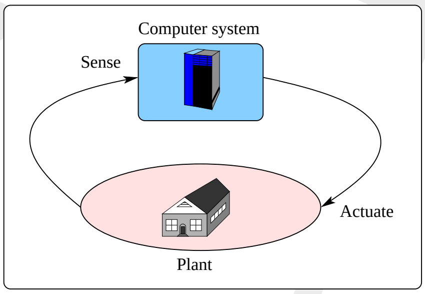


Figure 1.2: Closed loop control of a plant

We can picture this as in Figure 1.2, where our real-time computer system senses various things at the plant, and then uses these to control the operation of the plant by manipulating various actuators. Note that we use the term *plant* loosely here. It could refer to a factory, a car brake system, a rocket control system and so on.

Since the correct functioning of the computer system is dependant on external physical factors, for correct operation, the system time and external physical time should be the same, or at least they must have a predictable relationship. The computing system must sense, compute and actuate in a timely fashion. If the computer (for example) took too long to open a valve, then an important chemical may not be supplied to a tank, resulting in the temperature of the tank rising, resulting in an increase in pressure, resulting in an explosion (and so on and so on). We may have many sensors, and many actuators, and many (independant) control functions. Despite all this, for correct operation, we must schedule computational tasks for different control functions in a timely fashion.

1.2.1 Hard real-time systems

We can categorize the types of real-time systems by considering timing constraints:

Definition 2 *A **soft** real time system must meet timing constraints **often**.*

> An example of a system which might need this type of real-time system is in transaction processing, or for (say) multi-media streaming applications. It doesn't really matter if the music pauses for one-tenth of a second does it?

Definition 3 *A **hard** real time system must meet timing constraints **always**.*

> An example of a system which might need this type of real-time system is in Nuclear reactor control systems, flight control systems (fly-by-wire), and automotive electronics.

The hard/soft categorization is normally determined externally by the particular system we are constructing. Here we focus on the analysis of *hard* real-time systems, and will consider the following characteristics of hard real-time computing systems:

- ❖ Timeliness (The system should respond on time)
- ❖ Peak Load Handling (The system should not fail at peak load conditions)
- ❖ Predictability (The system should have totally predictable behaviour)
- ❖ Fault Tolerance (The system should tolerate failure of subsystems)
- ❖ Maintainability (The system should be easy to maintain and modify)

Some of these characteristics impact on the overall system architecture. In particular, we must avoid sources of nondeterminism. For example, if the time taken for software to respond to some event could be arbitrarily (nondeterministically) delayed, then we may be unable to guarantee some time constraint, and our system is no longer a *hard* real-time system.

Nondeterminism can arise in various ways. I am sure that we have all experienced unexpected and unexplained delays when using our computers. It is likely that most of these delays are linked to delays in systems that we do not consider part of our system. For example, a dead DNS server may give rise to quite long delays, as your computer attempts to use it, fails, and then uses a secondary one. In a global sense this is of course deterministic, but since we often do not consider the DNS heirarchy to be part of our computer system, we would classify this as locally nondeterministic.

Consider these sources of (local) nondeterminism:

- Direct Memory Access (DMA) by peripheral devices.
- Contention for system bus.
- Cache
- Interrupts generated by I/O devices.
- Memory Management (paging)
- Dynamic data structures, recursion, unbounded loops (language level).

Any one of these sources of nondeterminism could cause unexpected delays in a control system, and though such delays may be acceptable for the computer in a remote (infra-red) controller, they would not be acceptable in many other applications. Consider critical applications such as nuclear plant control systems; a failure here may result in unacceptable failures. We also rely on computers for railway switching systems, automotive applications, flight control and so on. For each item in this growing list of critical applications we expect a high degree of assurance that the controlling software is safe under all circumstances.

1.3 Steps towards assurance

The current state of affairs in the production of control software for hard real-time systems is dominated by a low level approach. The code is often written in assembly language, using programmable timers, direct handling of devices to ensure that they meet timing responses, and precise handling of interrupt and task priorities and behaviour. The goal of this approach is to achieve optimized predictable execution of code on simple (embedded system) computer architectures.

However, this approach has drawbacks. The programming is tedious and error prone, and the code quality depends on the particular skills of the programmer. The resultant code may be difficult to understand and maintain. The verification of timing constraints is almost impossible and so the system could collapse in rare and unforeseen circumstances leading to disasters.

In Buttazzo's book [BB97], the following laws of real-time systems are introduced:

- If something can go wrong, it will go wrong. (Murphys law)
- Any software bug will tend to maximize damage.
- The worst software bug will be discovered 6 months after the field test.
- A system will stop working at the worst possible time.
- Sooner or later the worst possible combinations of circumstances will occur.

The list seems to overstate the case, but the experience of computer and system engineers says otherwise. So, what can be done to improve software quality? The following points may help in improving the quality of delivered software:

- **Design software before building it:** You would not pay for a house without plans, why would you pay for software without similar sureties?
- **Reduce the complexity:** By modularity, well defined interfaces and so on.
- **Enforce compatibility with design:** Easier to say than to do

In each of the above points, we can use mechanical (or formal) methods. Other engineering disciplines have embraced formal methods[3], but the software industry (perhaps it should still be called a homecraft) has not yet to any great extent. Why is this? Perhaps software engineers are afraid of mathematically based methods, with a corresponding steep learning curve. Perhaps also there is widespread ignorance of the benefits, and a belief that there are few tools available.

The use of formal methods twenty-five years ago was seen as unworkable outside of the classroom, but now there are many well established cases of successful use of formal methods, resulting in products delivered under budget and under time[4].

[3] When you design a bridge or a multi-storey building, extensive stress and pressure analyses are performed. Hardware engineers regularly use simulators and testers for proposed designs. Most modern VHDL chips are designed using tools that automatically check and verify desired behaviour expressed using mathematical methods.

[4] There are also of course examples of products developed using formal methods delivered over

Hall [Hal90] and Bowen [BH95] outline the following *myths* relating to formal methods:

- ❖ Guarantee perfect software
- ❖ All about proving programs correct
- ❖ Only useful in safety critical systems
- ❖ Needs mathematicians
- ❖ Increases development cost
- ❖ Unacceptable to users
- ❖ Not used on large scale systems
- ❖ Delays development
- ❖ No tools available
- ❖ Mustn't be mixed with traditional methods
- ❖ Only applies to software
- ❖ Not needed
- ❖ Not supported
- ❖ FM people always use FM

The above myths are fully recognized as such these days, and formal methods are gradually becoming more common.

Formal software specification techniques are replacing flow charts and pseudo-code. The distinct advantages of formal specifications are:

1. The formal software specification can easily feed back modifications to the requirements specification as needed.
2. The formal software specification can be mechanically progressed through to a design and even implementation.
3. The formal software specification can be analysed, modelled and checked with a range of automated checkers.

The use of formal methods in industry is at least 25 years old. In some cases, products have been developed by competing teams of developers, one set using formal methods, and the other not. In these cases, the design phase took about 25% longer with the team using formal methods, but the overall development time was reduced (despite having to learn a new paradigm), and the resultant products were considered qualitatively better. In summary, formal techniques reduce the time to product delivery, and improve the products.

budget and over time, but in general these developments foundered due to other factors - not the 'use of formal methods'.

1.3.1 Assurance in hard real-time systems

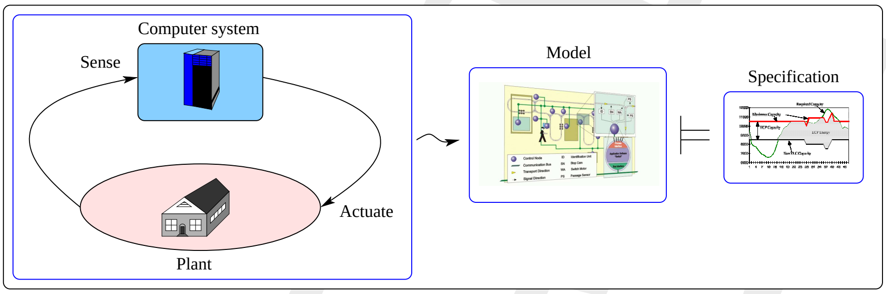


Figure 1.3: Modelling an entire system

Our overall goal is to improve the reliability of real-time systems. The closed system/verification approach in Figure 1.3 is to extract a model of the entire system (plant and controller), and then verify that this model meets some required specification. To use formal methods in hard real-time systems, we expect to be able to model the real-time systems precisely, capturing external and system events. We expect to be able to verify *timing* properties, verifying that an implementation meets a specification at every level.

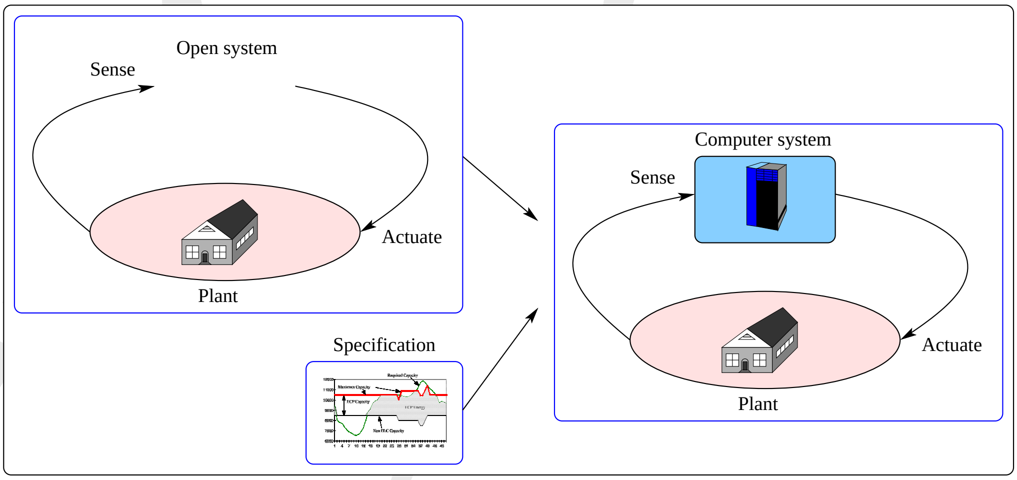


Figure 1.4: Synthesizing a controller

Another approach is that of synthesis outlined in Figure 1.4. Here the idea is that, given an open system, synthesize a controller that matches a given specification. Development of an RT system may involve both approaches.

What specific formal methods can be applied to hard real-time systems? We offer the use of *timed automata* to capture high-level descriptions of systems, and the use of *time-triggered architectures*, along with their formal analysis. A mix of these two paradigms can provide a high degree of assurance in the production of high-quality hard real-time systems.

1.3.2 Analysis of finite state systems

In the analysis of finite state systems, we consider the relationship between a program and its specification as relationships between automata. For such finite state systems, we consider the transitions from each state to another, and there are two ways for describing the behaviour:

1. An (infinite) path of state transitions. A logic that is interpreted on paths is called linear-time temporal logic (LTL).
2. An (infinite) tree, where branches in the tree correspond to (nondeterministic) choices of the system. A logic that is interpreted on these trees is called branching-time temporal logic.

There is a clear link (to be explored later) between linear-time temporal logics and finite state automata, and this forms a starting point for many system design methodologies, for example SDL, UML, POLIS. Verification tools such as SPIN, SMV are freely available, and have been used in many system developments.

We begin by considering the transition system (TS), a basic computational model, which can be used to model systems, and which can be checked for simple temporal properties (like A happened before B). Later we will develop this into timed automata, which can be used to verify temporal properties more precisely (like A happened at least 2 time units before B). A formal definition of a state transition system is:

Definition 4 *A state transition system* TS *is a 4-tuple* $(S, \mathrm{Act}, \Longrightarrow, S_{\mathrm{in}})$, *where*

1. S *is a set of* ***states***
2. Act *is a set of* ***actions***
3. $\Longrightarrow$: $S \times \mathrm{Act} \times S$ *is the* ***transition relation***
4. $S_{\mathrm{in}} \subseteq S$ *is the set of* ***initial states***

If you add *accepting conditions* to a state transition system, then the resultant machine is an automaton. In a state transition system, there may be more than one transition out of a given state, with the same input. Finite state transition systems can be represented as directed graphs.

Consider the system depicted in Figure 1.5, consisting of a train, a gate and a gate controller.

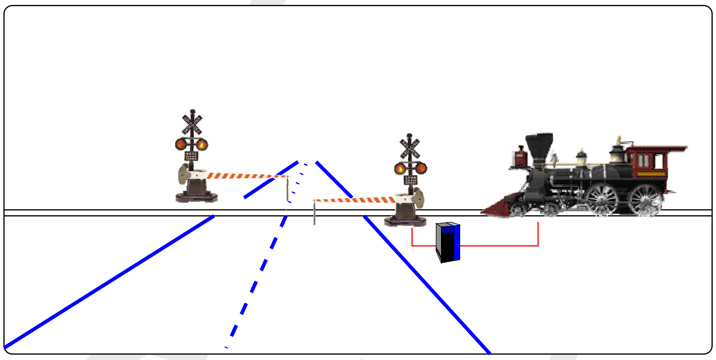
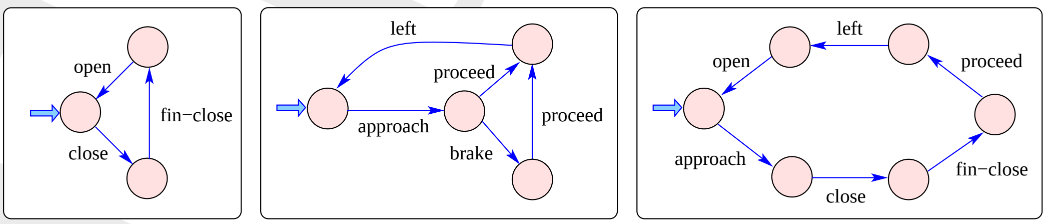


Figure 1.5: The train-crossing

As the train approaches, (sensed by a trackside sensor) the gate controller should lower the gate to prevent cars from crossing the line. We can model this by using three TSs, one for each of the components of the system.

There is an intuitive way of showing the corresponding TSs for the three components of this system, by using a directed graph, where the nodes represent the states, and labelled arcs represent the transitions from state to state. The labels on the arcs identify particular actions (signals or events) associated with each transition from one state to another. A starting node is identified by the $\Rightarrow$ arc.



(a) Gate (b) Train (c) Gate controller

Figure 1.6: Three automata for the train-crossing system

Figure 1.6 shows three TSs, each one representing one of the components. For example, the TS for the gate identifies the starting state, and if the gate receives a `close` signal, it moves to the next state, closing the gate. When the gate is closed, it signals with the `fin-close` signal, and then waits for the `open` signal to raise the crossing gate. The signals or events (formally, the *actions*) for each of the components of this system are shown in the following table.

Direction	**Gate**	**Train**	**Controller**
in	open close	proceed	approach fin-close left
out	fin-close	approach left brake	close proceed open

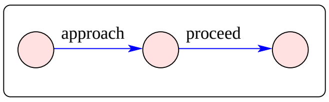
To model the whole system, we form a new transition system, the parallel composition of the train, gate and controller. Unfortunately, this model of the train-crossing system is not particularly useful, as it does not fit in with the reality of the train system; we do not know when to brake!

1.4 *Timed* automata

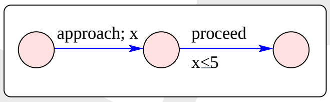
We would prefer (for example) to be able to specify timing information, and to be able to specify temporal constraints. For example, the following assertions specify some temporal constraints over the behaviour of the system:

- ❖ From the time `approach` is sent, `proceed` must be received within 5 time units.
- ❖ From the time `close` is sent, `fin-close` must be received within 3 time units.
- ❖ From the time `close` is received it takes at least 2 time units before `fin-close` is sent out.

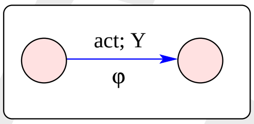
In order to capture timing information in a model, an approach is to add *clock* variables to transition systems, leading to systems that can model finite state control systems with timing constraints. The theory for these *timed automata* is rich, and practical tools are available. We will return to a more formal treatment later, but the underlying idea is to replace something like



... with ...



where the meaning attached is that "From the time `approach` is sent, `proceed` must be received within 5 time units".



We label the arcs with three annotations, described below:

1. act is an action (event/signal)
2. Y is a set of clock variables that are reset to 0.
3. φ is a clock constraint $\varphi ::= x \leq c \mid x \geq c \mid x < c \mid x > c \mid \varphi_1 \wedge \varphi_2$.

We can now construct more interesting *timed* automata for our system. For example, Figure 1.7 shows a new automata for the gate that signals that it needs to be repaired if it fails to close the gate after 4 time units.

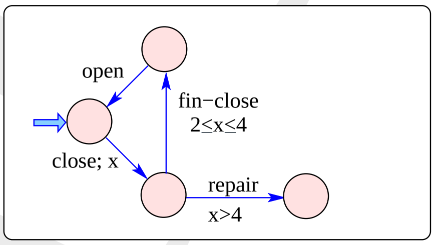


Figure 1.7: Timed automata for the gate

1.5 Time-triggered architectures

We are all familiar with the ethernet and wireless ethernet communication systems that, along with the TCP/IP network protocols, are used to inter-connect our computers. Such systems and protocols are considered much too high-level for interconnecting the small computers used to control car braking systems, steering systems and so on. The CAN (and the chips that support it) system was a response to this, providing hardware and simple software that could be used to interconnect lots of smaller computers, and even just simple sensors.

A CAN-enabled system responds to incoming messages, and we term this *event*-triggered. Event-triggered system architectures constructed from underlying communications systems like CAN may not be all that predictable in their behaviour, and for hard real-time systems, more precise temporal control is needed.

The time-triggered architecural paradigm [EBK] is a set of ideas for organizing real-time computing systems. In industries such as the automotive industry, there is a considerable need for highly dependable systems, and the paradigm provides a mechanism for achieving it. The architecture is now also used in by-wire systems in modern aeroplanes such as the AIRBUS 380. FlexRay is a closely related industry standard, supported by BMW, Daimler-Benz, Bosch and so on.

The main idea of a time-triggered architecture is that it provides time-slots for each component of a system to communicate. Consider a meeting of a large number of people, where everyone speaks into a microphone whenever he or she has something to say, with a rule that each speaker must wait for the current speaker to finish before starting. We might end up with all sorts of chaos, with speakers hogging the microphone, speakers colluding to exclude another speaker and so on.

In a time-triggered system, every speaker would be assigned a predetermined time slot. After one round, the speaker gets a slot again. In addition, a topic-schedule has been worked out in advance. Top1, Top2, Top4 in the first round. Top1, Top3 and Top5 in the second round Top2, Top4 and Top5 in the third round and so on. The time-triggered system ensures that no one breaks the rules.

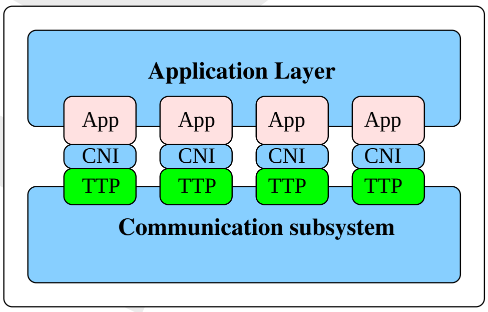


Figure 1.8: Time-triggered architecture

In Figure 1.8, we see 4 nodes, each running some application (App), and communicating with the others through an API termed the Communication Node Interface (CNI), which in turn is using a time-triggered protocol (TTP).

Each node may be a processor with its own I-O subsystem, operating system, the application software, and a time-triggered communication controller.

1.6 Summary of topics

In this section, we introduced the following topics:

Motivation. Some key ideas to motivate the study.
Real-time systems. Clarification of the hard real-time system.
Assurance. Steps to take to achieve assurance of the correctness of hard real-time systems.
Timed automata. An initial brief introduction to timed automata.
Time-triggered architecture. A brief introduction to time-triggered architectures.

1.7 Supplemental material

1.7.1 Exercises for Chapter 1

1. Develop a timed-automata diagram for the train.
2. Develop a timed-automata diagram for the gate controller.

1.7.2 Recommended reading

- Time-Triggered Paradigm [EBK]
  http://citeseer.ist.psu.edu/elmenreich03timetriggered.html