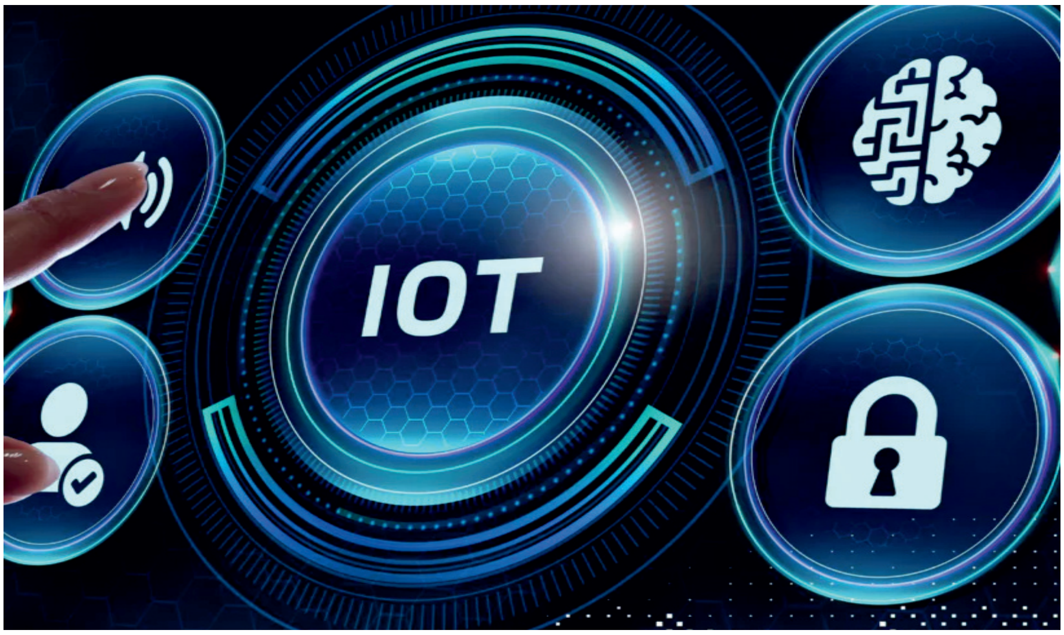
RECENT ADVANCES IN EMBEDDED SYSTEM



Akshatha K
Puneet Kalia

ALEXIS PRESS
JERSEY CITY, USA

RECENT ADVANCES IN EMBEDDED SYSTEM

RECENT ADVANCES IN EMBEDDED SYSTEM

Akshatha K

Puneet Kalia

ALEXIS PRESS

ALEXIS PRESS

Published by: Alexis Press, LLC, Jersey City, USA
www.alexispress.us

First Published 2022

A catalogue record for this publication is available from the British Library

Library of Congress Cataloguing in Publication Data

Includes bibliographical references and index.

Recent Advances in Embedded System by *Akshatha K, Puneet Kalia*

ISBN 978-1-64532-871-1

CONTENTS

CHAPTER 1

AN INTRODUCTION TO EMBEDDED SYSTEM

Ms. Akshatha K, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email Id- akshatha.k@presidencyuniversity.in

ABSTRACT:

An embedded system is a computer system that is designed to perform specific functions within a larger system. It typically consists of a microcontroller or microprocessor, memory, and other components that are integrated into a single device. Embedded systems are commonly found in a wide range of applications, including automotive, medical devices, consumer electronics, and industrial automation. Embedded systems is their ability to operate in real-time, meaning that they can respond to events and inputs in a timely and predictable manner. This is often accomplished through the use of specialized software and hardware components that are designed to handle specific tasks.

KEYWORDS:

Automation Consumer Electronics, Embedded System, Medical Device, microprocessor.

INTRODUCTION

Embedded systems are all around us, from the tiny microcontrollers powering our home appliances to the sophisticated control systems used in spacecraft. An embedded system is a combination of hardware and software designed to perform a specific function within a larger system. These systems are often designed to operate autonomously or with minimal human intervention and can be found in a wide variety of applications, from consumer electronics to medical devices to industrial automation.

The term "embedded" refers to the fact that the system is integrated into a larger device or system, and is typically not visible to the end-user. Embedded systems are often highly specialized, tailored to specific applications and optimized for performance, power consumption, and cost. They are designed to interact with the physical world and often include sensors, actuators, and other peripherals that allow them to interact with their environment . Embedded systems have been around for decades, but their importance has grown exponentially in recent years with the rise of the Internet of Things (IoT) and the proliferation of connected devices. As more and more devices become connected to the internet, the need for small, efficient, and powerful embedded systems has become more critical than ever[1], [2].

One of the defining characteristics of embedded systems is their real-time nature. Real-time systems are designed to respond to events as they happen, often in fractions of a second. This requires specialized hardware and software that can respond quickly and accurately to external stimuli. Real-time embedded systems are used in a wide variety of applications, from controlling the timing and synchronization of industrial machinery to managing the safety-critical functions of medical devices. Another important aspect of embedded systems is their low power consumption. Many embedded systems are battery-powered or otherwise limited in terms of power supply, which means that they must be designed to operate efficiently and conserve power whenever possible. This requires careful consideration of the hardware and software components of the system, as well as optimization of power management features[3], [4].

Embedded systems are typically designed by multidisciplinary teams of engineers, including hardware designers, software developers, and system architects. The design process can be complex and challenging, as it requires consideration of a wide range of factors, including performance, power consumption, cost, reliability, and security. Embedded system designers must also be familiar with a range of programming languages and software development tools, as well as hardware design techniques and principles. One of the key challenges facing embedded system designers is the need to balance the conflicting requirements of performance and power consumption. As devices become more sophisticated and powerful, the demand for increased performance has grown, but this must be balanced against the need to conserve power and extend battery life. This requires careful consideration of the hardware and software components of the system, as well as the algorithms and data structures used to implement the system's functionality[5], [6].

Another important consideration for embedded systems is security. As more and more devices become connected to the internet, the need for secure and reliable systems has become more critical than ever. Embedded systems are particularly vulnerable to attacks, as they often have limited resources and are designed to operate in challenging environments. Designers must take a range of security measures into consideration, including encryption, authentication, and access control, to ensure that their systems are protected against cyber threats.

Embedded systems are an integral part of our modern world and are critical to the functioning of many devices and systems. As devices become more connected and sophisticated, the demand for small, efficient, and powerful embedded systems has grown, and the importance of these systems is only set to increase in the years to come. Embedded system designers face a range of challenges, from balancing performance and power consumption to ensuring the security and reliability of their systems, but the rewards of developing successful embedded systems are significant, as they can have a significant impact on the way we live and work.

Embedded systems are used in a wide range of industries and applications. Some of the most common applications include consumer electronics, automotive systems, medical devices, industrial automation, and aerospace and defense. In consumer electronics, embedded systems are used in devices such as smartphones, smart home appliances, and gaming consoles. These systems are designed to provide fast, reliable, and intuitive user experiences while also conserving power and extending battery life.

In automotive systems, embedded systems are used in everything from engine management and control to safety-critical functions such as anti-lock brakes and airbag deployment. These systems must be designed to operate in harsh environments and withstand shocks, vibrations, and extreme temperatures. They must also be highly reliable and operate in real-time to respond to changing road and traffic conditions.

For embedded systems to be effective and durable, low-power hardware and software design is essential. In order to evaluate the energy performance of software and hardware, accurate energy consumption measurement is essential. This measurement also offers design insights that may be used to create new paradigms for algorithm or code optimization in terms of energy efficiency and system cost. There is some literature that has studied this subject area, but it lacks in-depth comparisons and analysis. We first demonstrate in this study the importance of precise energy consumption analysis for embedded systems. The main approaches for evaluating the energy use of embedded systems that have been studied in the literature may be divided into three categories: measurement-based energy profiling, model-based energy estimation, and simulator-based energy estimation. On the basis of these techniques' qualities, several subgroups are further created.

A crucial component of engineering education is offering students the opportunity to use what they learn in the classroom and to be exposed to real-world issues while they are still in school. One of the key courses that ties theoretical electrical engineering education to the actual world is the Embedded Systems course at Princess Sumaya University for Technology (PSUT). The use of project-based learning to improve the teaching of the Embedded Systems course at PSUT is described in this study. The comments from students demonstrated the method's success in improving students' comprehension and capacity for using embedded systems design ideas to address actual engineering challenges.

Many applications, including those for home automation, automobile systems, air surveillance, and, more recently, IoT subnets, typically use distributed embedded systems (Internet of things). Due to the variety of hardware and software as well as the differences in message flows that should occur based on protocol use and application flow requirements, distributed embedded systems are highly complicated. Hardware, software, and the technology used to connect the different embedded systems all need to be tested for the distributed embedded systems. Numerous devices, tools, techniques, and procedures are needed to test distributed embedded systems. It is difficult to test the communication that takes place between the various embedded systems. Comprehensive testing requires the continuous availability of the complete distributed embedded system, which cannot be guaranteed, as well as the testing system.

An agricultural nation's growth depends heavily on agriculture. Because the monsoon is unpredictable, unequal water is a significant issue all year long. All of this results in poor productivity and inadequate output. By increasing the effectiveness of agricultural technology, the agricultural sector's scientific approach will result in a fundamental shift in crop production. The architecture of practically all sectors has been altered by the Internet of Things (IoT), the quickening growth of technology, from statistics to quantitative approaches, and smart agriculture. This radical transformation, intended to upend the conventional wisdom, has brought up a number of difficulties as well as fresh possibilities. IoT and embedded system (ES) development and embedded sensors in the smart agricultural sector monitoring are offered as the basis for smart agriculture design.

Evaluation, group physical education, and data from a product's decision-making process may be used to compare the off-the-shelf standard. The use of Genetic Algorithm in actual education and preparation assistance for augmented reality and computer-generated reality. The application model of Virtual Reality (VR) innovation in this sector is created using real education and preparation. Highlights of VR innovation, for example, discernment, presence, intellect, self-control, and so forth, may be widely used in real education, and preparation plays a key role. All contemporary electronic components have embedded systems as a necessary component. In addition to being the foundation of the Internet, embedded systems, very specialized vertical markets, and Internet of Things solutions are employed in many network installations and network applications. Devices that are embedded are made to construct certain computer systems.

Evaluation, group physical education, and data from a product's decision-making process may be used to compare the off-the-shelf standard. The use of Genetic Algorithm in actual education and

preparation assistance for augmented reality and computer-generated reality. The application model of Virtual Reality (VR) innovation in this sector is created using real education and preparation. Highlights of VR innovation, for example, discernment, presence, intellect, self-control, and so forth, may be widely used in real education, and preparation plays a key role. All contemporary electronic components have embedded systems as a necessary component. In addition to being the foundation of the Internet, embedded systems, very specialised vertical markets, and Internet of Things solutions are employed in many network installations and network applications. Devices that are embedded are made to construct certain computer systems.

In order to engage and excite their pupils in the classroom, educators have joined a wider variety of non-specialists in the creation (design and development) of embedded systems. Because of this variety and the complexity of current embedded systems development platforms, there are prohibitive hurdles to entry. The rationale, specifications, implementation, and assessment of a novel programming platform that allows even inexperienced users to produce effective and efficient software for embedded devices are presented in this work. The platform consists of two main parts: CODAL, an effective component-oriented C++ runtime for microcontrollers, and Microsoft MakeCode, a web application that encompasses an accessible IDE for microcontrollers. We demonstrate how MakeCode and CODAL work together to provide an installation-free, cross-platform, high level programming experience for embedded devices without compromising performance or effectiveness.

Every digital device's central processing unit is called an embedded system. Understanding the embedded system is essential for the design and development of new digital devices and systems. Learning embedded systems in the context of engineering education is still difficult for millennials. Due to the fact that it requires the use of mathematics, programming, and practical understanding of the electronic and electrical components based in it. For new-generation learners to meet their need for understanding of embedded systems, traditional teaching technique has become less effective. As a result, new technical tools and methods must be included into the instructional methods. A new technology in the area of engineering education is augmented reality (AR). This technology has proliferated in the educational sector to simplify difficult topics via the use of computer-generated 3D data, animation, visual effects, and immersion. An augmented reality-based approach for teaching embedded systems to engineering students is presented in this research. To provide students with an AR learning experience, the suggested framework makes use of a physical user interface made up of AR markers, a USB camera, a display device, and the processing unit. The engineering education faculty examined the newly designed system's usefulness[7], [8].

DISCUSSION

Medical devices are another important application area for embedded systems. These systems are used in everything from pacemakers and insulin pumps to diagnostic equipment and surgical robots. They must be designed to operate safely and reliably in a wide range of environments, from hospitals to patients' homes. They must also be highly secure to protect patient data and prevent unauthorized access.In industrial automation, embedded systems are used in everything from factory automation to building control systems. These systems must be designed to operate in real-time and respond to changing conditions quickly and accurately. They must also be highly reliable and operate in harsh environments, with temperature extremes, dust, and humidity.

In aerospace and defense, embedded systems are used in everything from avionics and spacecraft control systems to missile guidance and navigation. These systems must be designed to operate in extreme environments, from the vacuum of space to high-temperature and high-pressure environments. They must also be highly secure to prevent unauthorized access and ensure the safety of personnel and equipment. The development of embedded systems is a complex process that requires a range of specialized skills and knowledge. The process typically begins with the identification of requirements, which are used to define the system's functionality and performance characteristics. This is followed by the design phase, where the system's hardware and software components are developed and integrated.

The design phase typically involves a range of tools and techniques, including computer-aided design (CAD), simulation, and modeling. These tools allow designers to develop and test different components of the system before they are integrated into the final product.Once the system's hardware and software components have been developed, they must be integrated and tested. This involves the use of specialized test equipment and techniques to ensure that the system meets its performance and reliability requirements. Testing is typically performed in a range of environments, from the lab to the field, to ensure that the system operates correctly under a wide range of conditions.

Once the system has been developed and tested, it must be deployed and maintained. This involves the installation of the system in its final environment and the provision of ongoing support and maintenance. This may include the provision of software updates, hardware repairs, and other maintenance activities to ensure that the system continues to operate correctly over its lifetime.One of the key challenges facing embedded system designers is the need to balance the conflicting requirements of performance and power consumption. As devices become more sophisticated and powerful, the demand for increased performance has grown, but this must be balanced against the need to conserve power and extend battery life. This requires careful consideration of the hardware and software components of the system, as well as the algorithms and data structures used to implement the system's functionality.

Another important consideration for embedded systems is security. As more and more devices become connected to the internet, the need for secure and reliable systems has become more critical than ever. Embedded systems are particularly vulnerable to attacks, as they often have limited resources and are designed to operate in challenging environments. Designers must take a range of security measures into consideration, including encryption, authentication, and access control, to ensure that their systems are protected.A cyber-physical system (CPS) combines computer with physical processes, and the behaviour of the system is determined by both its cyber and physical components. Physical processes are monitored and controlled by embedded computers and networks, often via feedback loops where calculations are affected by physical processes and vice versa.

Thus, CPS is about the meeting of the physical and the digital, not their unification. Separate understanding of the computational and physical components is insufficient. Instead, we must comprehend how they interact we describe the engineering concepts of such systems and the procedures by which they are created using a few CPS examples. The information technology (IT) revolution of the 20th century may be surpassed by CPS applications. Take a look at the following instances[7], [9].The heart must often be stopped during cardiac surgery in order to be operated on, and it must afterwards be restarted. Such a procedure is quite dangerous and has a lot of negative side effects. In place of stopping the heart, a surgeon might do surgery on a beating heart, according to a number of study teams. This is made feasible by two essential notions. First, surgical instruments may be robotically controlled to move in sync with heartbeat. Therefore, while the heart is still beating, a surgeon may use a gadget to continuously apply pressure to a specific place on the heart. Additionally, a stereoscopic video system may give the surgeon the illusion of a motionless heart via video (Rice, 2008). Although the heart is still beating, it seems to the surgeon that the heart has stopped. It takes comprehensive modelling of the heart, the equipment, the computer gear, and the software to produce such a surgical system. Software must be carefully designed to provide accurate timing and secure fallback procedures to manage errors. Additionally, in order to guarantee high confidence, a thorough investigation of the models and designs is required.

Instance 1.2 Imagine a city where automobiles and traffic lights work together to maintain a smooth flow of traffic. Imagine, for example, that you would never have to stop at a red light unless there was genuine traffic. A costly infrastructure that can identify automobiles on the road may be used to implement such a system. However, it could be preferable to have the automobiles themselves collaborate. They coordinate the usage of common resources, such as junctions, by keeping track of their location and communicating. Naturally, the feasibility of such a system depends on its reliability. Failures could have devastating results.

A well-designed flight control system may stop certain reasons of a crash even if it is impossible to stop all potential causes. These systems are effective illustrations of cyber-physical systems. In conventional aircraft, the pilot steers the craft using mechanical and hydraulic connections between cockpit controls and moveable surfaces on the wings and tail. In a fly-by-wire aircraft, the flight computer relays the pilot's directions to actuators in the wings and tail through an electrical network. Fly-by-wire aeroplanes use less fuel since they are lighter than conventional aircraft. Additionally, they have shown to be more dependable[10].Since a computer mediates the pilot's orders in a fly-by-wire aircraft, the computer may alter the pilot's commands. Many contemporary flight control systems alter pilot instructions under certain conditions. For instance, Airbus commercial aircraft utilise a method called flight envelope protection to stop an aircraft from straying outside of its safe operating range. For instance, they may stop a pilot from initiating a stall.

The idea of flying envelope protection may be expanded to assist avoid certain more accident causes. For instance, if implemented, Lee (2001)'s soft walls system would follow the position of the aircraft on which it is mounted and keep it from flying into terrain features like mountains and structures. In Lee's idea, the fly-by-wire flight control system generates a virtual pushing force that pushes the aeroplane away when it approaches the border of an obstacle. The pilot has a soft wall feeling that causes the plane to deviate.

The development and deployment of such a system faces several difficulties, both technological and non-technical. For a discussion of some of these concerns. Although the soft walls system from the preceding example seems somewhat far-fetched, there are more practical variations that have been implemented or are in advanced phases of research and development in the field of automobile safety. For instance, many modern vehicles alert the driver when lane changes are made accidentally. Think about the considerably harder issue of automatically changing the driver's behaviour. Clearly, this is more difficult than just alerting the motorist.

How can you be certain that the system will respond and take control only when necessary and in the precise amount that intervention is required? Numerous more uses come to mind, including those that help the elderly, enable telesurgery, which enables a surgeon to do an operation from a distance, and enable household appliances to work together to reduce the load on the power grid. Furthermore, it is simple to imagine how CPS could be used to enhance a variety of current systems, including robotic manufacturing systems, electric power generation and distribution, process control in chemical factories, distributed computer games, the transportation of manufactured goods, building heating, cooling, and lighting systems, elevators, and bridges that keep track of their own health. Such advancements have the potential to have a huge influence on economics, energy consumption, and safety.

Numerous of the aforementioned examples will be implemented using a framework similar to the one. This drawing is divided into three major sections. The "physical" component of a cyber-physical system is firstly the physical plant. Simply expressed, it refers to the component of the system that cannot be implemented using computers or digital networks. Mechanical components, biological or chemical processes, or human operators may all be a part of it. The second is one or more computational platforms, which may include one or more processors, actuators, operating systems, and sensors. A network fabric, which offers the means for the computers to interact, is the third component. The platforms and network fabric work together to create the "cyber" component of the cyber-physical system.

Data produced by the sensors via the physical plant is impacted by the actions done by the actuators. Actuator 1 on Platform 2 in the picture directs the physical plant. Sensor 2 is used to measure the physical plant's processes. The Computation 2 box implements a control rule that decides what instructions to provide the actuator depending on sensor data. A feedback control loop is the name given to such a loop. Platform 1 uses Sensor 1 to take further measurements and communicates with Platform 2 over the network fabric. A new control rule is realised in Computation 3 and combined with Computation 2's, perhaps preempting it.

When using a print-on-demand service, think about a high-speed printing press. This might have a similar organisational structure but with a lot more platforms, sensors, and actuators. The motors that press paper and apply ink on it may be controlled by the actuators. The regulations may provide a plan of action. At the US National Science Foundation in 2006, Helen Gill invented the phrase "cyber-physical systems." Although we may be inclined to equate the word "cyberspace" with CPS, its origins are earlier and deeper. Instead of seeing them as having descended from one another, it would be more correct to see the phrases "cyberspace" and "cyber-physical systems" as coming from the same root, "cybernetics."

Norbert Wiener (Wiener, 1948), an American mathematician who had a significant influence on the development of control systems theory, is credited with coining the term "cybernetics." Wiener invented the system that allowed anti-aircraft weapons to automatically aim and fire during World War II. Although he did not utilise digital computers, the techniques he used share many of the same concepts as the vast majority of computer-based feedback control systems in use today. According to Wiener, the word comes from the Greek letter "v" (kybernetes), which might refer to a helmsman, governor, pilot, or rudder.

The analogy applies well to control systems. Control and communication are combined in cybernetics, according to Wiener. His conception of control was firmly based on closed-loop feedback, in which measurements of physical processes drive the control logic, which in turn drives the physical processes. The combination of physical processes, calculation, and communication is known as cybernetics, even though Wiener did not employ digital computers; instead, control logic functions as a computation. Wiener was unable to foresee the significant impacts of digital networks and computing. The immense influence that CPS will have is further shown by the fact that the term "cyber-physical systems" is ambiguously defined as the combination of cyberspace with physical processes. CPS uses information technology that is significantly superior to even the most optimistic predictions made in Wiener's day.

Internet of Things (IoT), Industry 4.0, the Industrial Internet, Machine-to-Machine (M2M), the Internet of Everything, TSensors (trillion sensors), and the Fog are all phrases that are now in vogue (like the Cloud, but closer to the ground). All of them illustrate a technology that profoundly links our physical and digital worlds. Since the word CPS does not specifically relate to either implementation strategies (such as the "Internet" in IoT) or specific applications (such as the "Industry" in Industry 4.0), we believe it to be more enduring and fundamental than any of the others. Instead, it concentrates on the basic philosophical conundrum of combining the physical and cybernetic engineering traditions depending on the kind of paper, the temperature, and the humidity for adjusting for paper stretch. In the event of a paper jam, a networked system similar may be utilized to trigger a quick shutdown in order to protect the equipment. To avoid calamities, such shutdowns must be carefully planned throughout the whole system. High-end instrumentation systems and the generation and distribution of energy experience comparable circumstances.

We provide a compelling example of a cyber-physical system in this section. The purpose of this example is to highlight the significance of the wide range of subjects discussed in this literature. The particular application is the Stanford Testbed of Autonomous Rotorcraft for Multi Agent Control (STARMAC), created in collaboration between Stanford and Berkeley by Claire Tomlin and colleagues (Hoffmann et al., 2004). The STARMAC, a tiny quadrotor aircraft, in flight. Its main objective is to act as a testbed for 6 Introduction to Embedded Systems, Lee & Seshia working with methods for multi-vehicle autonomous control. The goal is to enable multi-vehicle cooperation for a shared job.

The creation of such a system is fraught with difficulties. The first is that driving the car is not easy. The four rotors, which provide a changeable amount of downward force, are the major actuators. The vehicle's four rotors' propulsion may be balanced to allow it to take off, land, spin, and even flip in the air. How do we decide how much thrust to use? The vehicle's weight is another important factor. It naturally becomes heavier as it gets heavier because it has to transport more stored energy. It requires more force to fly the heavier it is, which calls for larger and more powerful motors and rotors. When the car is so hefty that the rotors endanger people, the design has crossed a critical barrier. Safety is a major issue, even with a very small vehicle, hence the system must be developed with fault management.

Thirdly, the vehicle must function in a context and engage with its surroundings. It may, for instance, be continuously run by a vigilant person using a remote control. Or, it may be anticipated to function autonomously, to take off, do a task, go back, and land.

The Design Processing is not able to profit from the alert human's observation. Greater sensor sophistication is necessary for autonomous operation. The vehicle must do localisation in order to maintain track of its whereabouts. It must be able to detect obstructions and recognise the location of the ground. Such vehicles may even be able to land on a ship's pitching deck by themselves with appropriate design. Additionally, the vehicle must constantly check its own performance in order to identify problems and respond appropriately in order to minimise damage. It is simple to envision several alternative applications that have characteristics in common with the quadrotor challenge. Operating on a beating heart is analogous to the challenge of landing a quadrotor craft on a tossing ship. It needs a thorough knowledge of the interplay between the dynamics of the embedded system the quadrotor, the robot, and the dynamics of the environment the ship, the heart.

Using the quadrotor example to show how the different pieces contribute to the design of such a system, the remainder of this chapter will describe the various components of this book. Understanding how to build and execute cyber-physical systems is the aim of this book. Modeling, design, and analysis are the three main steps in the process. Modeling is the practise of studying a system more thoroughly via imitation. Models reflect and copy the characteristics of the system. Models outline what a system does. Design is the methodical production of objects. It describes how a system carries out its functions. Analysis is the process of dissecting a system in order to understand it better. It explains the rationale behind a system's actions or fails to do what a model says it should do. These three stages of the process overlap, and the design process repeatedly switches between them. Modeling is often the first step in the process, with the aim of comprehending the issue and creating potential solutions. In Figure 1 illustrate the embedded system.

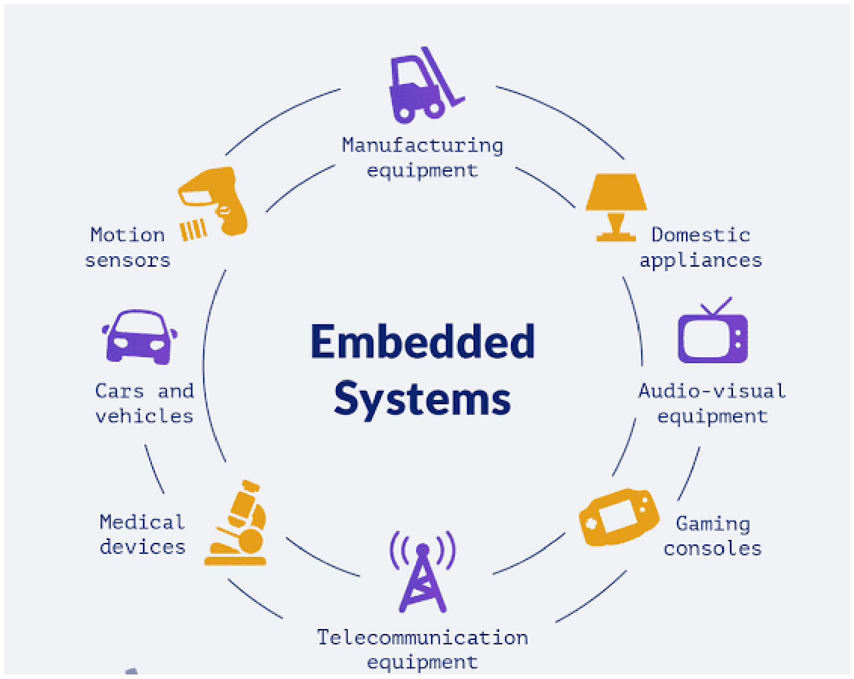


Figure 1: Shows the embedded system.

To start solving the quad rotor issue, we may build models that convert human inputs to move laterally or vertically into instructions to the four motors to generate thrust. According to a model, the vehicle would tilt and move laterally if the force on all four rotors is different. Then, it would construct state machines that represent the modes of operation, such as takeoff, landing, hovering, and lateral flight, using methods similar (Discrete Dynamics). The system could then combine these two kinds of models using the methods (Hybrid Systems), resulting in hybrid system models of the system that could be used to examine the changes between operating modes. Following that, mechanisms for composing models of multiple vehicles, models of interactions between a vehicle and its environment, and models of interactions among components within a vehicle would be provided by the techniques (Composition of State Machines) and 6 (Concurrent Models of Computation) systems, operating systems, wireless networks, etc.). The STARMAC architecture (reproduced with permission). When a first prototype reveals errors in the models, the models may need to be revised and taken back to the modelling stage.

A number of sensors are located at the left and bottom of the image, which the vehicle uses to locate itself and learn about its surroundings. Three boxes in the center display three different microprocessors. The low-level control algorithms necessary to keep the vehicle flying are carried out by the Robostix, an Atmel AVR 8-bit microcontroller that works without an operating system. Higher-level activities are carried out by the other two processors with the aid of an operating system. Both processors are equipped with wireless communications that ground controllers and cooperative vehicles may utilise. Analysis plays a significant role early on in a healthy design process. Models and designs will be subjected to analysis. The models may be examined for safety requirements, such as ensuring an invariant that states that a vehicle's vertical speed should not be more than 0.1 metres per second if it is within one metre of the ground. The designs may be examined for the software's timing behaviour, such as how quickly the system reacts to an emergency shutdown instruction. Details of both models and designs will be involved in certain analytical difficulties. Understanding how the system will react in the quadrotor example if network connection is lost and the vehicle cannot be reached is crucial. How does the car know when contact is lost? Accurate software and network modelling will be necessary for this.

We would next check that these safety features are met by software implementations using the methods (Equivalence and Refinement) and (Reachability Analysis and Model Checking). The methods (Quantitative Analysis) would be used to check if the programme complies with real-time restrictions approaches would be used to guarantee that the quadrotor cannot be taken over by bad parties and that whatever private information it may be acquiring is not disclosed to an enemy. The book's first section, on modelling, focuses on models of dynamic behaviour. The major topic of modelling physical dynamics is briefly covered with an emphasis on continuous dynamics in time which also emphasises how important it is for designers to understand the semantics of composition. Concurrent models of computation are covered (Concurrent Models of Computation), including many of the ones found in Simulink and LabVIEW, two popular design tools used by practitioners.

In the chapter on modelling, we describe a system as only a collection of components that are taken into account collectively. In contrast to conceptual or logical systems like software and algorithms, a physical system is one that is actualized in substance. A system's dynamics is how it changes through time, or how it evolves. A physical system model is a description of certain features of the system that aims to shed light on some of the system's characteristics. The models used in this article contain mathematical characteristics that make systematic analysis possible. Since the model replicates the system's characteristics, it provides understanding of the system.

A model is a system unto itself. It's crucial to distinguish between a model and the system it represents. There are two separate artefacts here. If a model of a system precisely captures the characteristics of the system, it is said to have high fidelity. If specifics are left out, it is called to abstract the system. Physical system models are always abstractions of the real system because they must always leave out some aspects. Developing a grasp of how to utilise models, how to take use of their strengths, and how to respect their flaws is one of this text's main objectives.

A cyber-physical system (CPS) is a system that combines computer, networking, and physical components. All three components are often included in models of cyber-physical systems. Typically, the models must capture both dynamical and static features (those that do not change during the operation of the system). It's vital to remember that a cyber-physical system model doesn't always need to incorporate both discrete and continuous components. A model that is exclusively discrete (or exclusively continuous) may have high fidelity for the desired features.

It would take more than one chapter or even one book to cover all of the modelling approaches covered in this section of the book. Such models are in fact the subject of 12 Introduction to Embedded Systems, Lee & Seshia several areas of biology, physics, chemistry, and engineering. Our strategy is geared for engineers. We begin by assuming that the reader has some familiarity with mathematical dynamics modelling (calculus courses that include physics examples are adequate), and then we concentrate on building a variety of models. The combined modelling of the cyber side, which is logical and conceptual, with the physical side, which is embodied in matter, will constitute the problem's core for the cyber-physical system. We thus avoid attempting to be exhaustive in favour of selecting a small number of well-known and often used modelling approaches, reviewing them, and then combining them to create a cyber-physical whole.

The book's second section, which reflects the subject's inherent variability, has a totally different tone. This section focuses on embedded system design with a particular emphasis on their function inside a CPS (Sensors and Actuators), with a focus on how to describe them so that their function in the dynamics of the whole system is understood. The topic of processor architectures is covered in Chapter 8 (Embedded Processors), with specific attention paid to those aspects that make embedded systems work best. Memory architectures (Memory Architectures), which also covers physical characteristics like memory technologies, architectural characteristics like memory hierarchy (caches, scratchpads, etc.), and abstractions like memory models in programming languages.

The impact of memory design on dynamics is the main focus. The digital/analog interface, including sampling, and input/output techniques in software and computer systems are covered. The concepts that underpin operating systems (Multitasking), with a focus on multitasking. The reader is intended to be persuaded that there is genuine benefit in applying the modelling approaches presented in the first part of the book by emphasising the drawbacks of using low-level processes like threads. These modelling methods assist designers in developing trust in system designs. Real-time scheduling is covered (Scheduling), which also introduces the topic.

Given the importance of concurrency and time control in the design of cyber-physical systems, we pay close attention to these processes in all of the design-related chapters. Embedded processors often have a specific purpose when used in a product. They can manage an automobile engine or gauge Arctic ice thickness. Increasing their specialization may have a significant positive impact. For instance, they could use a lot less power and so can be used for extended periods of time with few batteries. Or they could have specialist equipment to carry out tasks like image analysis that would be expensive to carry out on standard gear. The reader should be able to critically assess the many technological options accessible with the help of this section.

Teaching students to construct systems while thinking beyond conventional abstraction levels, such as hardware and software, computation and physical processes, is one of the objectives of this section of the book. Although this kind of cross-layer thinking is helpful when constructing systems generally, it is especially crucial in embedded systems due to their diverse nature. For instance, in order to construct a control method that is stated in terms of real-valued values, a programmer must have a firm grasp of computer arithmetic (for instance, of fixed-point numbers).

Similar to this, a developer of automotive software that must adhere to real-time limitations must be aware of processor capabilities like pipelines and caches that may influence how quickly tasks are completed and, therefore, how the system behaves in real-time. Similar to this, an implementer of interrupt-driven or multi-threaded software must be aware of the atomic operations made available by the underlying software-hardware platform and apply the proper synchronization mechanisms to guarantee accuracy. This section of the book employs homework tasks to help readers get a better knowledge of these cross-layer concepts rather than doing an extensive study of various implementation approaches and platforms.

CONCLUSION

Embedded systems are a critical component of modern technology and play a vital role in a wide range of industries and applications. From consumer electronics to aerospace and defense, embedded systems are used to provide fast, reliable, and intuitive user experiences while also conserving power and extending battery life. The development of embedded systems is a complex process that requires a range of specialized skills and knowledge, from the identification of requirements to the deployment and maintenance of the final product.

REFERENCES:

[1] S. Gauhar Fatima, S. K. Fatima, S. Abdul Sattar, S. Adil, and K. F. Ahmed, “Home automation using Zigbee technology and IoT,” *Int. J. Adv. Res. Eng. Technol.*, 2019, doi: 10.34218/IJARET.10.2.2019.011.

[2] M. N. Kapadia, P. D. Sonawane, and A. Marhatta, “Experimentation & analysis of door sag for refrigerator,” *Int. J. Recent Technol. Eng.*, 2019, doi: 10.35940/ijrte.B3034.078219.

[3] M. Agrawal, J. Zhou, and D. Chang, “A survey on lightweight authenticated encryption and challenges for securing industrial IoT,” in *Advanced Sciences and Technologies for Security Applications*, 2019. doi: 10.1007/978-3-030-12330-7_4.

[4] T. Tumurbaatar and T. Kim, “Comparative study of relative-pose estimations from a

monocular image sequence in computer vision and photogrammetry," *Sensors (Switzerland)*, 2019, doi: 10.3390/s19081905.

[5] Khin Nyein Win | Lwin Lwin Htay | Nyan Phyo Aung, "Automatic Water Storage and Distribution System using Reliance SCADA," *Int. J. Trend Sci. Res. Dev.*, 2019, doi: https://doi.org/10.31142/ijtsrd26414.

[6] A. Ahmad, "Reliable and Fault Tolerant Systems on Chip Through Design for Testability," in *Proceedings - 2019 Amity International Conference on Artificial Intelligence, AICAI 2019*, 2019. doi: 10.1109/AICAI.2019.8701390.

[7] L. A. Dobrzański and A. D. Dobrzańska-Danikiewicz, "Why are carbon-based materials important in civilization progress and especially in the industry 4.0 stage of the industrial revolution," *Materials Performance and Characterization*. 2019. doi: 10.1520/MPC20190145.

[8] S. Eom, R. M. Voyles, and D. Kusuma, "Embedding intelligence into smart tupperware brings internet of things home," in *Annual Technical Conference - ANTEC, Conference Proceedings*, 2019.

[9] A. Jain, O. Prakash, A. Talukdar, S. P. Samal, R. Nanjundappa, and N. Mahesha, "Screen intelligence engine: ML based method for predicting IoT devices using screen contents," in *2019 IEEE 16th India Council International Conference, INDICON 2019 - Symposium Proceedings*, 2019. doi: 10.1109/INDICON47234.2019.9030306.

[10] E. Bouzekri *et al.*, "Engineering issues related to the development of a recommender system in a critical context: Application to interactive cockpits," *Int. J. Hum. Comput. Stud.*, 2019, doi: 10.1016/j.ijhcs.2018.05.001.

CHAPTER 2

A COMPREHENSIVE STUDY ON CONTINUOUS DYNAMICS

Ms. Natya S, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email Id- natyas@presidencyuniversity.in

ABSTRACT:

Continuous dynamics is a branch of mathematics and physics that deals with the study of systems that evolve continuously over time. These systems are described by mathematical models that involve differential equations and partial differential equations, and they arise in various fields, such as mechanics, biology, economics, and engineering. The study of continuous dynamics involves the investigation of the behavior of these systems over time, including their stability, periodicity, and chaos. It also involves the analysis of their long-term behavior and the study of the effects of external perturbations on the system. Continuous dynamics has many applications in science and engineering, such as the design and control of physical systems, the analysis of ecological and economic systems, and the understanding of biological and physiological processes. It is a rich and active field of research, and it continues to provide new insights into the behavior of complex systems in the natural world.

KEYWORDS:

Biology, Continuous Dynamics, Economics, Mechanics, Periodicity.

INTRODUCTION

Continuous dynamics is a term used in physics and mathematics to describe the behavior of systems that change continuously over time. Unlike discrete dynamics, which involves systems that change in steps or jumps, continuous dynamics involves systems that change smoothly and continuously. Continuous dynamics is a fundamental concept in the study of complex systems, and it has numerous applications in fields such as physics, engineering, biology, and economics. In physics, continuous dynamics is used to describe the motion of objects and the behavior of physical systems, such as fluid flow and electromagnetic fields. In engineering, continuous dynamics is used to model and simulate the behavior of complex systems, such as control systems and manufacturing processes. In biology, continuous dynamics is used to model the behavior of biological systems, such as the growth and development of organisms. And in economics, continuous dynamics is used to model the behavior of markets and economic systems [1].

One of the key concepts in continuous dynamics is the idea of a differential equation. A differential equation is an equation that relates the rate of change of a system to its current state. In other words, it describes how the system changes over time based on its current state. Differential equations can be used to model a wide range of physical, biological, and economic systems, and they are an essential tool in the study of continuous dynamics. Another important concept in continuous dynamics is the notion of a phase space. A phase space is a mathematical representation of a system that captures all possible states of the system at a given point in time. In other words, it is a space that describes the state of the system in terms of its relevant variables. For example, in physics, the phase space of a particle might include its position and velocity. In economics, the phase space of a market might include the price and quantity of a particular commodity [2].

One of the most important ideas in continuous dynamics is the concept of attractors. An attractor is a set of states in the phase space of a system that the system tends to move towards over time. Attractors can be either stable or unstable, depending on whether small perturbations from the attractor lead the system back towards the attractor or away from it. Attractors are a key concept in the study of chaos theory, which is the study of complex, nonlinear systems that exhibit unpredictable behavior.

Another important concept in continuous dynamics is the idea of bifurcations. A bifurcation is a point in the phase space of a system where the behavior of the system changes dramatically. For example, in the case of a pendulum, a bifurcation might occur when the pendulum changes from swinging back and forth to spinning around in circles. Bifurcations are often associated with the emergence of new behavior in a system and can be used to study the stability and resilience of a system [3].

The study of continuous dynamics has led to numerous important discoveries and insights in physics, biology, economics, and many other fields. For example, in physics, the study of fluid dynamics has led to a better understanding of the behavior of fluids and the design of more efficient engines and turbines. In biology, the study of continuous dynamics has led to a better understanding of the growth and development of organisms and the spread of diseases. And in economics, the study of continuous dynamics has led to a better understanding of the behavior of markets and the factors that drive economic growth.

In conclusion, continuous dynamics is a fundamental concept in the study of complex systems that change continuously over time. It is a powerful tool that can be used to model and simulate the behavior of physical, biological, and economic systems, and it has numerous applications in a wide range of fields. By studying continuous dynamics, scientists and engineers can gain new insights into the behavior of complex systems and develop new technologies and strategies for solving some of the world's most pressing problems.

Continuous dynamics is a field of study that explores how systems change over time in a continuous and smooth manner. It is a fundamental concept in many fields of science, such as physics, engineering, biology, and economics. Continuous dynamics is based on differential equations, which relate the rate of change of a system to its current state. The study of continuous dynamics involves concepts such as phase spaces, attractors, bifurcations, and chaos theory [4].

One important application of continuous dynamics is in the field of fluid dynamics. Fluids are complex systems that change continuously over time, and the study of fluid dynamics is crucial for understanding the behavior of fluids and designing more efficient engines, turbines, and other fluid-based systems. In fluid dynamics, continuous dynamics is used to model the behavior of fluids and predict how they will behave under different conditions. For example, differential equations can be used to model the flow of water through a pipe, and phase spaces can be used to represent the different states of the water in the pipe.

Another important application of continuous dynamics is in the field of control systems. Control systems are systems that are designed to control the behavior of other systems, such as manufacturing processes, robots, and aircraft. Continuous dynamics is used to model and simulate the behavior of control systems and to design control strategies that are effective and efficient. For example, differential equations can be used to model the behavior of a robot arm, and phase spaces can be used to represent the different states of the arm.

In biology, continuous dynamics is used to model the growth and development of organisms. Biological systems are complex systems that change continuously over time, and the study of continuous dynamics is crucial for understanding how organisms grow and develop. Differential equations can be used to model the growth of a population of bacteria, and phase spaces can be used to represent the different states of the bacteria.

LITERATURE REVIEW

Michael J. Spivey et al. The best way to characterise real-time cognition is not as a series of logical operations carried out on discrete symbols but rather as a dynamic pattern of neural activity. The consistency of these dynamics shows that mental activity does not lend itself to the language labels relied on by most of psychology when it occurs between states of consciousness that may be described. We examine eye-tracking and mouse-tracking evidence for this temporal continuity and give geometric visualisations of mental activity, portraying it as a continuous course through a state space (a multidimensional space in which locations correspond to mental states). However, the majority of the mental trajectory is in intermediate regions of that space, revealing graded mixtures of mental states. When the state of the system travels towards a frequently visited region of that space, the destination may constitute recognition of a specific word or a specific object [5].

Jan Lorenz et al. Deffuant et al. They have attracted a good deal of interest from the complexity science, sociophysics, and social simulation groups. Researchers from fields including physics, mathematics, computer science, social psychology, and philosophy are working on it. In these models, agents have ongoing views that they can progressively change if they learn what other people think. According to the theory of bounded confidence, agents can only communicate with one another if their opinions are similar. The models may also be recast based on the agent's density in the opinion space in a master equation style. Typically, the models are examined through agent-based simulations in a Monte Carlo style. The fourfold benefit of this survey. The frameworks for agent-based and density-based modelling, which include examples of multidimensional opinions and heterogeneous boundaries of confidence, will be shown first. It will also provide the cluster configuration bifurcation diagrams for the homogeneous model with evenly distributed starting beliefs [6].

Seung Yeal Ha et al. We assume that the simulation findings for the discrete dynamical systems correspond to what is occurring in the equivalent continuous dynamical systems since numerical simulations for continuous dynamical systems are based on their discretized models in time and space. This is generally true for any limited time frame given appropriate system parameter circumstances. With the help of exponential synchronisation estimates and a tight confinement of the initial phase to a small subset of the state space, we demonstrate in this paper a smooth transition from discrete dynamics to continuous dynamics for the ensemble of Kuramoto oscillators that is valid over the entire time interval. For the discrete Kuramoto model and its enhanced second-order extension, we carry out this uniform-in-time transition [7].

Yuji Yamamoto et al. From the viewpoint of a dynamical system, complex human behaviour, such as interlimb and interpersonal coordination, has been investigated. We examine how a dynamical system approach which incorporates continuous, discrete, and switching dynamics can be applied to a sports event. Using a relative phase analysis, it was discovered that during expert kendo bouts, switching between in- and anti-phase synchronisation was regulated by an interpersonal distance of 0.1 metres. Return map analysis was used on the time series of movements during kendo bouts in the discrete dynamical system. Attractors and repellers, or six coordination patterns, were used to categorise offensive and defensive actions. Furthermore, these attractors and repellers displayed two distinct states. The coordination patterns and switching behaviour were then elucidated by calculating state transition probabilities based on the two states. We proposed switching dynamics with temporal inputs to elucidate the fundamental principles underlying the complicated behaviour related to switching inputs in a striking action as a non-autonomous system [8].

Pinzeng Rao et al. The use of remote sensing data for continuous, high spatiotemporal-resolution dynamic monitoring of lake water extents is still constrained by the pervasiveness of noise, such as clouds and cloud shadows. The goal of this work is to develop a method for mapping continuously occurring time series of very precise lake water extents. Four lakes were chosen as instances from various parts of China. Two sets of MODIS products, including MOD09A1 and MOD13Q1, are utilised to extract water bodies in order to minimise the effects of noise and guarantee high spatial and temporal resolution of the final findings. This strategy primarily consists of data fusion, post processing, and preliminary categorization. The first classification results were effectively and automatically obtained using the Random Forest (RF) classifier. Post-processing is used to fix the noise-affected classification findings as much as feasible. [9].

Glenn J. Martyna et al. Unfortunately, for tiny or rigid systems, the Nosé-Hoover dynamics is not ergodic. Here, a change to the dynamics is suggested that uses a Nosé-Hoover chain instead of a single thermostat variable. The canonical distribution is provided by the "new" dynamics, when the basic formalism fails. In addition, the new technique is simpler to use than an extension which likewise produces the canonical distribution for stiff instances. The Homologous Data-Based Spatial and Temporal Adaptive Fusion Method (HDSTAFM), which lessens the impact of noise and also improves the temporal and spatial resolution for the final water results, is used to fuse the processed results of the two sets of products. Landsat-based water data were used to measure accuracy, and the values for overall accuracy (OA), user's accuracy (UA), producer's accuracy (PA), and kappa coefficients (KC) are often more than 0.9. A time series of water area and altimetry data collected by several satellites, as well as water-level data chosen from hydrological stations, both showed good agreement [10].

Denis Stijepic et al. We go through the continuous dynamics models on the 2-simplex that result from applying various qualitative constraints on the (continuous) functions that produce the dynamics on the 2-simplex. We take into account three different categories of qualitative constraints: inequality conditions (or set-theoretical conditions), monotonicity/curvature conditions (or differential-geometrical conditions), and topological conditions (which refer to (transversal) non-(self-) intersection of trajectories). We go through the effects of these limitations on transitional and limit dynamics on the 2-simplex as well as the many prospective and actual uses of the resultant system-theoretical models in economics, and in particular in economic growth and development theory [11].

DISCUSSION

Chaos theory is another important aspect of continuous dynamics. Chaos theory is the study of complex, nonlinear systems that exhibit unpredictable behavior. Chaotic systems are characterized by their sensitivity to initial conditions, which means that small changes in the initial conditions can lead to large differences in the behavior of the system over time. Chaos theory has applications in many fields, such as meteorology, economics, and biology. For example, chaotic behavior can be observed in the weather, the stock market, and the behavior of the heart.

Attractors are another important concept in continuous dynamics. An attractor is a set of states in the phase space of a system that the system tends to move towards over time. Attractors can be either stable or unstable, depending on whether small perturbations from the attractor lead the system back towards the attractor or away from it. Attractors are a key concept in the study of chaos theory, as they can help to explain the behavior of chaotic systems. Attractors also have applications in many fields, such as in the design of control systems, where they can be used to stabilize the behavior of a system.

Bifurcations are another important concept in continuous dynamics. A bifurcation is a point in the phase space of a system where the behavior of the system changes dramatically. For example, in the case of a pendulum, a bifurcation might occur when the pendulum changes from swinging back and forth to spinning around in circles. Bifurcations are often associated with the emergence of new behavior in a system and can be used to study the stability and resilience of a system.

Continuous dynamics is a powerful tool for understanding the behavior of complex systems. By studying continuous dynamics, scientists and engineers can gain new insights into the behavior of physical, biological, and economic systems and develop new technologies and strategies for solving some of the world's most pressing problems. For example, the study of continuous dynamics has led to the development

A few of the several modelling approaches for examining the dynamics of a physical system are reviewed in this chapter. We start by investigating moving mechanical components (classical mechanics). The methods utilised to research these components' dynamics generally to a wide range of different physical systems, including as circuits, chemical processes, and biological processes. However, as mechanical components are the most easily seen by most people, they provide our example concreteness. Differential equations, or their counterpart, integral equations, are often used to simulate the motion of mechanical components. These models are only truly effective for "smooth" motion, which may be defined more precisely using the concepts of linearity, temporal invariance, and continuity.

For movements that are not smooth, such as those depicting collisions of mechanical components, we may utilise modal models that represent. Mechanical item collisions may be effectively described as discrete, instantaneous occurrences. Hybrid systems modelling, which entails simultaneously simulating smooth motion and such discrete occurrences, is covered in Chapter 4. We are one step closer to combined modelling of cyber and physical processes thanks to such mixtures of discrete and continuous behaviours.

We start with basic equations of motion that provide the form of ordinary differential equations, modelling the system (ODEs). Then, we demonstrate how these ODEs may be represented in actor models, a class of models found in well-known modelling languages like Simulink (from The MathWorks, Inc.) and LabVIEW (from National Instruments). Then, we analyse the characteristics of these models, such as linearity, temporal invariance, and stability, and we consider how these characteristics affect model manipulation. We provide a straightforward illustration of a feedback control system that grounds an unsteady system. Such systems are a prime example of a cyber-physical system since software is often used to implement their controllers. The characteristics of the cyber and physical components influence those of the entire system. In Figure 1 shows the embedded control system design and simulation.

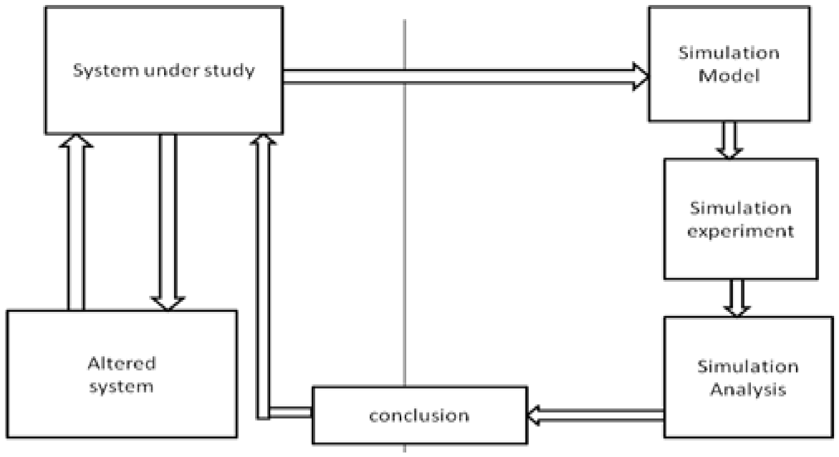


Figure 1:This shows the Embedded Control System Design.

This is not meant to be exhaustive; rather, it is meant to be just enough to allow you to build fascinating models. Many outstanding books on classical mechanics are recommended to the interested reader, including Goldstein (1980), Landau and Lifshitz (1976), Marion and Thornton, and others (1995). Physical objects' motion in space may be described using six degrees of freedom. Three of them indicate three-dimensional position, and three of them represent spatial orientation. By convention, x is drawn increasing to the right, y is drawn rising up, and z is drawn increasing out of the page.

Three axes, x, y, and z, are assumed. Roll θx is an angle of rotation about the x axis, where by convention an angle of 0 radians denotes horizontally flat along the z axis (i.e., the angle is provided relative to the z axis). The rotation around the y-axis is known as yaw, and according to convention, an angle of 0 radians is equivalent to pointing straight to the right (i.e., the angle is provided relative to the x-axis). Rotation around the z axis is referred to as pitch, and 0 radians is used to denote horizontal pointing in this context (i.e., the angle is given relative to the x axis).

Therefore, six functions of the type f: R R, where the domain is time and the codomain is either distance along an axis or angle with respect to an axis, may be used to describe the location of an object in space. Newton's second law, which links force with acceleration, governs changes in position or orientation. The second derivative of position is acceleration. The position information is handled by our first equation, F(t) = Mx (t), in which F is the force vector in three directions, M is the object's mass, and x is x's second derivative with respect to time (i.e., the acceleration). Velocity is essential.

2A continuous-time signal may only have a linked subset of R as its domain, such as R+, the non-negative reals, or the range from 0 to 1 inclusive. Although the codomain may be any set, real numbers are the most effective for describing physical quantities x(t) = x(0) + Z t 0 x ()d = x(0) + tx (0) + 1 M Z t 0 Z 0 F()dd, where x(0) is the starting position. Position is the integral of velocity. You may calculate an object's acceleration, velocity, and position at any time using these equations if you know its beginning location, initial velocity, and the forces acting on it in all three directions as a function of time.

Torque, the rotating equivalent of force, is used in the versions of these equations of motion that alter orientation. The net rotational force acting on an item is represented as a three-element vector as a function of time. Similar to equation (2.1), T(t) = d dt I(t) (t), (2.2), where T is the torque vector in three directions and I(t) is the object's moment of inertia tensor, it may be connected to angular velocity. The object's shape and orientation determine the 3 3 matrix that represents the moment of inertia. Intuitively, it describes the reluctance that an item has to spin about any axis as a function of its orientation along the three axes. If the object is spherical, for example, this reluctance is the same along all axes, hence it simplifies to a \sconstant scalar I (or equivalently, to a diagonal matrix I with equal diagonal elements I).

For instance, in this case, Iyx(t) is the inertia that governs how acceleration around the x axis is connected to torque around the y axis and Ty(t) is the net torque around the y axis (which would induce changes in yaw). Rotational velocity is the integral of acceleration, where (0) is the starting rotational velocity in three axes. Rotational velocity is expressed as (t) = (0) + Z t 0 () d. Using (2.3), this becomes (t) = ((0) + 1 I Z t 0 T(d) for a spherical object.

The rotational velocity's integral formula for orientation is (t) = (0) + Z t 0 ()d = (0) + t (0) + 1 I Z t 0 Z 0 T()dd, where (0) is the starting orientation. These equations allow you to calculate the rotational acceleration, velocity, and orientation of an item at any moment provided you know it beginning orientation, initial rotational velocity, and the torques acting on it in all three axes as a function of time.

Sometimes we may simplify by lowering the number of dimensions that are taken into account, as we have done with a spherical object. Such a simplification is often referred to as a model-order reduction. For instance, there may not be much need to take into account the object's pitch or roll if it is a moving vehicle on a level surface. Consider a simple control issue that allows for such dimensionality reduction. A helicopter has two rotors: one up top that gives it lift, and one on the tail that gives it motion. The helicopter's body would spin if the tail rotor weren't there.

That spin is countered by the rotor on the tail. In particular, the main rotor's torque must be balanced by the force generated by the tail rotor. Here, we examine the tail rotor's function apart from the helicopter's other motions. There is no need to take equations defining position into consideration since we are assuming that the helicopter's location is fixed at the origin. Additionally, because we believe the helicopter would stay vertical, pitch and roll are set at zero. Since we can specify the coordinate system to be fixed to the helicopter, these assumptions are not as implausible as they may seem.

According to these presumptions, the moment of inertia transforms into a scalar that symbolises a torque that prevents yaw variations. The variations in yaw will be due to Newton's third law, the action-reaction law, which asserts that every action has an equal \sand opposite response. As a result, the helicopter will probably start to spin anticlockwise to the direction of the rotor. In order to prevent the body of the helicopter from spinning, the tail rotor's role is to counteract that torque.

A system that receives as input a continuous-time signal Ty, the torque around the y axis, is used to represent the simplified helicopter (which causes changes in yaw). This Torque is the result of adding the forces produced by the main and tail rotors. That total equals 0 when things are fully balanced. Our system will produce the angular velocity y around the y axis as its output. It is possible to write (2.2) in dimensionally reduced form as y(t) = Ty(t)/Iyy. The crucial note about this example is that if we were to choose to describe the helicopter by, say, having x: R → R \s3 \srepresent the absolute location in space of the tail \sof the helicopter, we would end up with a significantly more sophisticated model. The control system's design would likewise be significantly more challenging.

A differential or integral equation that connects input signals (force or torque) to output signals (position, orientation, velocity, or rotational velocity) provides a model of a physical system in the preceding section. Such a physical system may be considered as a component in a bigger system. In specifically, a box having an input port and an output port may be used to mimic a continuous-time system, which uses continuous-time signals:

Here, the codomain denotes the signal's value at a certain moment, while the domain denotes time. If we want to explicitly represent a system that exists and begins to function at a certain moment in time, we may swap out the domain R with R+, the non-negative reals. The collection of functions that translate the reals into the reals, such as x and y above, is what makes up the system's model, and it has the form S: X Y, (2.5), where X = Y = R R.

The parameters of the system may affect how the function S behaves, in which case they may be optionally indicated in the box and optionally written in the function notation. If there are parameters p and q in the aforementioned figure, for instance, the system function may be written as Sp,q or even S(p,q), bearing in mind that both notations reflect functions of the type in 2.5. An actor is a box like the one in the example above, where the inputs are functions and the outputs are functions.

Both the input and the output are functions with continuous time. The actor's starting angular velocity (y(0)) and moment of inertia (Iyy) are its parameters. The following criteria characterise the actor's role. Models of actors may be constructed. Particularly, given two actors S1 and S2, we may create the following cascade composition: The set of all functions with domain R and codomain R is denoted by the notation R R, which may alternatively be written (R R). The "wire" connecting the output of S1 and the input of S2 signifies exactly that y1 = x2, or more pedantically, \s∀ t ∈ R, y1(t) = x2 (t).

We may express this equation more succinctly by writing y1 = ax1, where it is clear that the result of a scalar and a function x1 is equivalent. The right actor is a representation of an integrator parameterized by starting value I as described by t R, y2(t) = I + Z t 0 x2(d)d.

We can see that this system reflects, where the input x1 = Ty is torque and the output y2 = y is angular velocity, if we set the parameter values to a = 1/Iyy and I = y(0). The boxes that depict the actors in the above illustration are our own iconography. Assigning those recognized visual notations is helpful since these specific actors (scaler and integrator) are especially effective building blocks for constructing models of physical dynamics.

Actors with multiple input signals and/or multiple output signals are a possibility. Similar representations of these may be seen in the example below, which includes two input signals and one output signal: A signal adder is a very helpful construction piece with this shape, defined by t R, y(t) = x1(t) + x2 (t). When one of the inputs is sometimes subtracted rather than added, the icon is further modified to include a negative sign next to that input, as seen below: A function S is represented by this actor: Given by (R R), (S(x1, x2))(t) = y(t) = x1(t) x2 and (t R), (R R) 2 (R R) (t).

Unless the difference is crucial to the argument, we won't distinguish between a system and its actor model in the remaining sections of this chapter. We'll assume that the actor model adequately represents all of the system's interesting aspects. This is a big assertion, to be sure. The actor model's attributes often only provide rough descriptions of the real system. In this part, we examine some possible characteristics of actors and the systems they comprise, such as causality, memorylessness, linearity, temporal invariance, and stability. A system is logically considered causal if its output is solely dependent on its recent and previous inputs. It is a little difficult to make this idea exact, however. We do this by first noting "present and prior inputs." Take into account the continuous-time signal x: R A for some set A.

Let x|t represent the limitation in time function, which is defined exclusively for times t , and when it is defined, x|t (t) = x. (t). Therefore, x|t is the "present and previous inputs" at time t if x is an input to a system. Consider a continuous-time system S: X Y, where for some sets A and B, X = AR and Y = BR. If x1|t = x2|t and S(x1)|t = S(x2)|t for all x1, x2, X, and R, then this system is causal. In other words, the system is causal if the outputs are similar up to (and including) time for two alternative inputs x1 and x2 that are identical up to (and including) time. So far, we have explored just causal systems. If x1|t = x2|t and S(x1)|t = S(x2)|t are true for every x1, x2, X, and R, then the system is strictly causal. In other words, the system is strictly causal if the outputs are similar up to (and including) time for two alternative inputs x1 and x2 that are identical up to (and not including) time. Only previous inputs have any bearing. Of course, a strictly causal system is also causal. The Integrator actor only has causal effects. Although not precisely causal, the adder is still causal. Construction of feedback systems may benefit from strictly causal agents.

It makes sense that a system has memory if its output is dependent on both previous and present inputs (and future inputs, if the system is not causal). Consider a continuous-time system S: X Y, where for some sets A and B, X = AR and Y = BR. Formally, a system is considered memoryless if a function f: A B exists such that (S(x))(t) = f(x(t)) for all t R for all x X. That is, the input x(t) at time t is the sole factor that influences the output (S(x))(t) at time t. The adder is memoryless, although the integrator discussed above is not. Exercise 2 demonstrates that a system's output is constant for all inputs if it is strictly causal and memoryless. LTI (linear and time invariant) systems offer several excellent mathematical characteristics. On these principles, a large portion of control systems theory is predicated. These characteristics make up the bulk of signals and systems courses and are beyond the purview of this work. However, there are times when we will employ simplified versions of the attributes, therefore it is important to know when a system is LTI.

A system S: X → Y, where X and Y are sets of signals, is linear if it fulfils the superposition property: \s∀ x1, x2 ∈ X and ∀ a, b ∈ R, S(ax1 + bx2) = aS(x1) + bS (x2). The helicopter system shown in Example 2.1 may be seen to be linear if and only if the initial rotational velocity y(0) =

0. More broadly, it is clear that the Scale actor is always linear, the integrator specified in Example 2.3 is linear if and only if the starting value of I = 0, and that the cascade of any two linear actors is linear. The adder is also linear as we can easily expand the concept of linearity to actors with more than one input or output signal. We first create a specific kind of continuous-time actor called a delay before defining time invariance. Let D: X Y be defined by x X and t R, with (D: (x))(t) = x(t), where X and Y are sets of continuous-time signals. (2.7). Here, the delay actor's parameter is used. If x X and r, then a system S: X Y is time invariant and S(D(x)) = D(x)).

Time invariance does not apply to the helicopter system described. Time invariant, however, is a slight variant: y(t) = 1 Iyy Z t Ty()d. A system that is both linear and time invariant is referred to as an LTI. Selecting an LTI model whenever feasible is one of the main goals of modelling physical dynamics. It is worthwhile to do this approximation if it yields an LTI model in a decent manner. Finding models for which the approximation is acceptable or determining if the approximation is reasonable are not necessarily simple tasks. It is often simple to create models that are more complex than.

If the output signal is bounded for all bounded input signals, a system is said to be bounded-input bounded-output stable (BIBO stable or simply stable). Think of a continuous-time system that has an input of w and an output of v. If there is a real integer A such that |w(t)| A for every t R, then the input is said to be bounded. If there is a real integer B such that |v(t)| B for every t R, the output is said to be bounded. The system is stable if for each input limited by some A, there is some bound B on the output by t R, u(t) = 0, t 0 1, and t 0 (2.8) This indicates that previous to time zero, there is no torque supplied to the system, and \sstarting at time zero, we apply a torque of unit magnitude. This input's bounds are obvious. It never reaches one in magnitude. But the production continues to increase unabatedly. In real life, a helicopter employs a feedback mechanism to decide how much torque to apply to the tail rotor to maintain a straight body. Then, we research how to achieve it.

In a system with feedback, an actor's output is sent back to effect their own input, creating directed cycles. In most control systems, feedback is used. They measure an error e in the picture, which is the difference between the intended behaviour and the behaviour that actually occurs (y in the figure), and then utilise that measurement to change the behaviour. The feedback error measurement should be compensated for by the matching corrective signal (Ty in the figure) to lower subsequent errors. A feedback system must typically contain at least one strictly causal actor (the Helicopter in the diagram) in every directed cycle since the correction signal typically can only effect future mistakes.

Feedback control is a complex subject that may easily fill many volumes and whole courses. Here, we simply obliquely mention it in order to illustrate how software and physical systems interact. Embedded software is often used to construct feedback control systems, and the combination of physical dynamics and software results in total physical dynamics. The discrepancy between the intended and actual angular velocities is represented by the error signal e. In the illustration, the controller just scales the error signal by a constant K to provide the helicopter a control input. We write y(t) = y(0) + 1 Iyy Z t 0 Ty()d using equation (2.4), and y(t) = y(0) + K Iyy Z t 0 (() y())d using equation (2.9), where we utilised the facts (from the picture), e(t) = (t) y(t), and Ty(t) = Ke (t).

Equation (2.10) is difficult to answer since y(t) is present on both sides of the equation. Laplace transforms are the most straightforward solution method (see Lee and Varaiya (2011), Chapter 14). However, for our needs here, we may employ a more brute-force strategy from calculus. We assume that (t) = 0 for every t in order to keep things as simple as possible; that is, our goal in controlling the helicopter is to prevent it from spinning at all. Zero angular velocity is the preferred value. In this instance, (2.10) may be expressed as y(t) = y(0) K Iyy Z t 0 y()d. (2.11) \s32 Introduction to Embedded Systems, Lee & Seshia

Calculus teaches us that, for t 0, Z t 0 aea d = e atu(t) 1, where u is provided by (2.8), thus we can deduce that the answer to (2.11) is y(t) = y(0)e Kt/Iyy u. (t). As we can see from (2.12), as long as K is positive, the angular velocity decreases as t increases and approaches the target angular velocity (zero). The K will get closer more rapidly for bigger K. In an unstable system, angular velocity will increase unrestricted for negative K.The prior instance serves as an example of a proportional control feedback loop. It is termed thus because the control signal is proportionate to the mistake. We took the intended signal to be zero. In the following example, we establish the behaviour for a certain non-zero desired signal by assuming that the helicopter is initially at rest (the angular velocity is 0). Assume that the intended signal is (t) = au(t) for any constant a, and that the helicopter is initially at rest, with (0) = 0. In other words, we want to be able to regulate the helicopter such that it rotates at a certain speed.

Once again, when t increases and K is positive, the angular velocity increases until it reaches the required angular velocity. The K will get closer more rapidly for bigger K. In an unstable system, angular velocity will increase unrestricted for negative K. Notice how the required angular velocity is perfectly represented by the first element in the solution above. The second term is an error termed the tracking error that for this case asymptotically approaches zero.The aforementioned illustration is rather implausible since we are unable to independently regulate the helicopter's net torque. In particular, the net torque Ty is equal to the total of the torques produced by the top and tail rotors, or Tt and Tr, respectively. Quite independently from the rotation of the helicopter, Tt will be defined by the rotation necessary to maintain or reach a particular height. In order to stabilise the helicopter for any Tt, we will really need to construct a control system that regulates Tr (or, to be more accurate, 34 Lee & Seshia, Introduction to Embedded Systems

Tt is now viewed as an external (uncontrolled) input signal, and the feedback control system is only regulating Tr. How will the control system function? Again, a thorough examination of the topic is beyond the purview of this work, but we will focus on a single instance. Assume that Tt = bu(t), where bu is a constant, represents the torque generated by the top rotor. In other words, the top rotor begins spinning at a consistent speed at time zero and maintains that speed throughout. Let's assume that the helicopter starts off at rest. In the beginning, we change the model into the corresponding model this transformation only depends on the algebraic fact that Ka1 + a2 = K(a1 + a2/K) for any real integer a1, a2, and K.

CONCLUSION

The control system asymptotically approaches a non-zero angular rotation of b/K, although zero angular rotation is the ideal state. By raising the control system feedback gain K, this tracking error may be made arbitrarily tiny, but with this controller architecture, it cannot be made to go to zero studies a different controller architecture that results in an asymptotic tracking error of zero.

REFERENCES:

[1] T. Johnson, A. Kerzhner, C. J. J. Paredis, and R. Burkhart, “Integrating models and simulations of continuous dynamics into SysML,” *J. Comput. Inf. Sci. Eng.*, 2012, doi: 10.1115/1.4005452.

[2] T. J. Faulkenberry, M. Witte, and M. Hartmann, “Tracking the continuous dynamics of numerical processing: A brief review and editorial,” *J. Numer. Cogn.*, 2018, doi: 10.5964/jnc.v4i2.179.

[3] S. Tajima and R. Kanai, “Integrated information and dimensionality in continuous attractor dynamics,” *Neurosci. Conscious.*, 2017, doi: 10.1093/nc/nix011.

[4] F. Ghanbarnejad and K. Klemm, “Stability of Boolean and continuous dynamics,” *Phys. Rev. Lett.*, 2011, doi: 10.1103/PhysRevLett.107.188701.

[5] M. J. Spivey and R. Dale, “Continuous dynamics in real-time cognition,” *Curr. Dir. Psychol. Sci.*, 2006, doi: 10.1111/j.1467-8721.2006.00437.x.

[6] J. Lorenz, “Continuous opinion dynamics under bounded confidence: A survey,” *Int. J. Mod. Phys. C*, 2007, doi: 10.1142/S0129183107011789.

[7] S. Y. Ha, D. Kim, J. Kim, and X. Zhang, “Uniform-in-time transition from discrete to continuous dynamics in the Kuramoto synchronization,” *J. Math. Phys.*, 2019, doi: 10.1063/1.5051788.

[8] Y. Yamamoto, A. Kijima, M. Okumura, K. Yokoyama, and K. Gohara, “A switching hybrid dynamical system: Toward understanding complex interpersonal behavior,” *Applied Sciences (Switzerland)*. 2018. doi: 10.3390/app9010039.

[9] P. Rao, L. Lou, W. Jiang, X. Cao, Y. Wang, and X. Wang, “Continuous dynamics monitoring of multi-lake water extent using a spatial and temporal adaptive fusion method based on two sets of MODIS products,” *Sensors (Switzerland)*, 2019, doi: 10.3390/s19224873.

[10] G. J. Martyna, M. L. Klein, and M. Tuckerman, “Nosé-Hoover chains: The canonical ensemble via continuous dynamics,” *J. Chem. Phys.*, 1992, doi: 10.1063/1.463940.

[11] D. Stijepic, “Models of continuous dynamics on the 2-simplex and applications in economics,” *Adv. Syst. Sci. Appl.*, 2019, doi: 10.25728/assa.2019.19.1.591.

CHAPTER 3

DISCRETE DYNAMICS OF COMPLEX SYSTEMS

Mrs. Annapurna H. S., Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email Id- annapurna.hs@presidencyuniversity.in

ABSTRACT:

Discrete dynamics is the study of the behavior of discrete systems, which are those that change over time in a series of distinct steps. In mathematics, discrete dynamical systems are often represented by difference equations, which describe how the system evolves from one time step to the next. These systems are found in a wide range of fields, including physics, chemistry, and biology, economics, and computer science. The study of discrete dynamics involves analyzing the long-term behavior of these systems. This can include studying the stability and convergence of the system, identifying any periodic or chaotic behavior, and exploring the existence of fixed points or attractors.

KEYWORDS:

Biology, Chemistry, Chaotic Behavior, Dynamics, Economics.

INTRODUCTION

Discrete dynamics is a branch of mathematics that deals with the study of dynamical systems where time is represented by a sequence of discrete values, rather than a continuous variable. It is a subfield of dynamical systems theory, which focuses on the qualitative behavior of systems over time. Discrete dynamics is used to model and analyze a wide range of phenomena, including the behavior of populations, the evolution of biological systems, the behavior of financial markets, and the behavior of digital circuits.The study of discrete dynamics is often motivated by the observation that many natural and man-made systems exhibit discrete behavior. For example, the population of a species is a discrete quantity that changes over time as new individuals are born and old ones die. The behavior of digital circuits is also inherently discrete, as the signals are either on or off, and the circuits can only change state at discrete time intervals [1].

Discrete dynamical systems are defined by a set of rules that describe how the state of the system evolves over time. In the simplest case, a discrete dynamical system can be described by a single map that takes the current state of the system as input and produces the next state as output. This map can be viewed as a function that maps the state of the system at one time to its state at the next time. In more complex systems, there may be multiple maps that describe how different aspects of the system evolve over time.

The behavior of a discrete dynamical system is typically analyzed by studying the long-term behavior of the system, such as its stability, periodicity, or convergence to a fixed point or limit cycle. Stability analysis involves determining whether small perturbations to the systems initial state cause the system to converge to a fixed point or limit cycle, or whether they cause the system to diverge or exhibit chaotic behavior. Periodicity analysis involves determining whether the system's behavior exhibits repeating patterns over time. Convergence analysis involves determining whether the system approaches a particular state or set of states over time [2].

One of the most important concepts in discrete dynamics of a fixed point, which is a state of the system that does not change over time. A fixed point is said to be stable if small perturbations to the system's initial state cause the system to converge to the fixed point, and unstable if small perturbations cause the system to diverge away from the fixed point. Fixed points can be found by solving the equation f(x) = x, where f is the map that describes the evolution of the system. If the derivative at the fixed point is less than 1, the fixed point is stable, while if the derivative is greater than 1, the fixed point is unstable.

Another important concept in discrete dynamics of a limit cycle, which is a sequence of states that the system cycles through over time. A limit cycle can be stable or unstable, and can be analyzed using the same techniques as fixed points. The period of a limit cycle is the number of time steps it takes for the system to return to its initial state. A limit cycle with a period of 1 is called a fixed point. One of the key tools for analyzing discrete dynamical systems is the use of bifurcation diagrams, which show how the long-term behavior of the system changes as a parameter of the system is varied. Bifurcation diagrams can be used to identify regions of parameter space where the system exhibits different types of behavior, such as stable fixed points, unstable fixed points, or limit cycles [3].

Discrete dynamics is used in a wide range of applications, including biology, physics, economics, and computer science. In biology, discrete dynamics is used to model the behavior of populations, the spread of diseases, and the evolution of biological systems such as spin systems and lattice models. In economics, discrete dynamics is used to model the behavior of financial markets, the evolution of consumer preferences, and the dynamics of supply and demand. In computer science, discrete dynamics is used to model and analyze the behavior of digital circuits, software systems, and communication networks.

One important area of application for discrete dynamics is in the study of chaos theory. Chaos theory is the study of complex and unpredictable behavior in deterministic systems, and is an area of research that has gained significant attention in recent decades. Discrete dynamical systems are often used to model chaotic behavior, and can be used to explore the properties of chaotic systems in a controlled and mathematical framework. Another important area of application for discrete dynamics is in the field of control theory, which is the study of how to design systems that can control the behavior of other systems. Discrete dynamical systems are often used to model the behavior of control systems, and can be used to design controllers that can stabilize unstable systems, track desired trajectories, and minimize errors.

Discrete dynamics is a branch of mathematics that deals with the study of dynamical systems where time is represented by a sequence of discrete values. Discrete dynamics is used to model and analyze a wide range of natural and man-made systems, and is an important tool for understanding the behavior of these systems over time. The study of discrete dynamics is motivated by the observation that many systems exhibit discrete behavior, and can be analyzed using a range of techniques, including stability analysis, periodicity analysis, and convergence analysis. Discrete dynamics has applications in many fields, including biology, physics, economics, and computer science, and is an active area of research with many open questions and opportunities for further exploration [4].

One of the key concepts in discrete dynamics is the notion of chaos. Chaos theory is the study of how small changes in the initial conditions of a system can led to large and unpredictable changes in its long-term behavior. Discrete dynamical systems are often used to model chaotic behavior, and can exhibit a range of complex behaviors, including period-doubling bifurcations, strange attractors, and sensitive dependence on initial conditions. The study of chaotic systems has applications in many fields, including weather forecasting, fluid dynamics, and financial forecasting. Another important concept in discrete dynamics is that of nonlinear dynamics study of systems that exhibit nonlinear behavior. Nonlinear systems are systems where the output is not proportional to the input, and can exhibit a wide range of complex and unpredictable behaviors. Nonlinear dynamics is a rich and active area of research, with applications in fields such as control theory, signal processing, and communications.

Discrete dynamics is often used to model and analyze biological systems, including the behavior of populations, the spread of diseases, and the evolution of genetic systems. In these systems, discrete dynamics is used to model the discrete changes that occur over time, such as births and deaths, mutations, and changes in genetic frequencies. Discrete dynamics is also used to model the behavior of ecological systems, such as food webs and ecosystems, and can be used to analyze the stability and resilience of these systems to perturbations and disturbances.

Discrete dynamics is also used in physics, particularly in the study of spin systems and lattice models. Spin systems are physical systems where the behavior of each particle depends on the behavior of its neighboring particles, and can be used to model a wide range of physical phenomena, including magnetism and superconductivity. Lattice models are discrete models that describe the behavior of physical systems on a lattice, and can be used to study the behavior of condensed matter systems and the behavior of complex networks.

In computer science, discrete dynamics is used to model and analyze the behavior of digital circuits, software systems, and communication networks. Digital circuits are inherently discrete systems, and the behavior of these systems can be analyzed using techniques from discrete dynamics. Software systems are often modeled as discrete dynamical systems, and can be analyzed using techniques from formal methods and verification. Communication networks are also modeled as discrete systems, and the behavior of these systems can be analyzed using techniques from network theory and discrete dynamics.

Discrete dynamics is a powerful and versatile tool for modeling and analyzing a wide range of natural and man-made systems. The study of discrete dynamics is motivated by the observation that many systems exhibit discrete behavior, and can be used to explore a wide range of phenomena, including chaos, nonlinear dynamics, and the behavior of biological and physical systems. Discrete dynamics has applications in many fields, including biology, physics, economics, and computer science, and is an active area of research with many open questions and opportunities for further exploration.

LITERATURE REVIEW

J. P. Garcia-Sandoval et al. It is shown that the distribution of prime numbers may be studied using fractal theory in number theory. In order to visualize the recursive and symmetric properties of prime numbers, a set of discrete dynamics are proposed. These discrete dynamics are based on the similarities between the distribution of prime numbers' powers for each natural number and generalized Cantor's sets, and they are used to approximate a fractal version of the distribution function of prime numbers. Then, using this collection of discrete dynamics, a prime numbers sieve algorithm is suggested, and its fractal dimension is calculated, suggesting that the fractal and chaotic behavior on the distribution of prime numbers results from: 1) the expansion of the discrete dynamics' memory or dimension, which introduces new oscillation frequencies at each stage; and 2 the symmetry break caused by the exponential growth of the length of the discrete dynamics. Since the distribution of prime numbers follows the traditional road to chaos that is, from periodic to quasi periodic to chaos it exhibits deterministic chaotic behavior with symmetries and harmonics [5].

Alon Malka-Markovitz et al. an important thermally triggered mechanism in discrete dislocation dynamics (DDD) simulations of metal deformation. With little help from molecular dynamics (MD) simulations, we constructed a stress-dependent line-tension model in DDD simulations. This model enables the probabilistic cross-slip rate estimated in MD simulations for Cu across a wide range of loads and temperatures to be replicated in DDD simulations. In DDD simulations, the use of an atomically precise cross-slip model enables more realistic modelling of processes such deformation softening, dislocation-precipitate interaction, and dislocation patterning.

Alejandro Cabrera et al. the investigation of discrete groups' behaviour on linked manifolds in relation to the orbital spaces that are assumed to be differentiable stacks. We demonstrate how the dynamics up to conjugation and inversion are encoded in the orbit stack of a discrete dynamical system on a simply linked manifold. We also provide a generalisation of this conclusion for non-simply connected manifolds and arbitrary discrete groups, and we tie it to the covering theory of stacks. Applications include computing the stack-theoretic basic group of hyperbolic toral automorphisms, revisiting the classification of lens spaces, and obtaining a geometric interpretation of Rieffel's theorem on irrational rotations of the circle [6].

Inagaki, Hidehiko K. et al. Short-term memories connect disparate occurrences in time, such as a previous feeling and a subsequent action. Slow neuronal dynamics, such as selective sustained activity that may last for seconds, are associated with short-term memory. Neurons in the mouse anterior lateral motor cortex (ALM) display persistent activity that directs future movements in a delayed response task requiring short-term memory. Here, we coupled intracellular and extracellular electrophysiological with optogenetic manipulations and network modelling to ascertain the principles underlying this persistent activity. We demonstrate that the activity of ALM neurons shifted throughout the delay period in the direction of discrete end points that matched certain movement orientations. These end points were unaffected by brief changes in ALM activity brought on by optogenetic disturbances. Occasionally, perturbations altered the dynamics of the population to the opposite end point, which was followed by inappropriate actions. Our findings demonstrate that short-term memory linked to motor planning is underpinned by discrete attractor dynamics [7].

Sin Quek et al. In polycrystalline materials, grain boundaries (GBs) often play a significant role in preventing dislocations from gliding, leading to the well-known Hall-Petch effect. Simulations using molecular dynamics that show how nanocrystalline materials deform show that GBs perform substantially more. We enhance the now canonical discrete dislocation dynamics (DDD) modelling technique to account for GB sliding and the absorption, emission and transmission of lattice dislocations at GBs. GB dislocations are nucleated and migrate along the GB in a way that is an extension of the DDD formalism in the framework used to do this. We show that incorporating a dislocation model of GB dynamics enables all of these effects to cooperatively relax localised stress fields (such as those caused by dislocation pileups) and act synergistically to alter the mechanical response of polycrystals, going well beyond GBs merely blocking dislocation slip [8].

Feng Ding et al. Interest in the precise prediction of RNA's three-dimensional (3D) structure and folding dynamics has been rekindled by RNA molecules with unique activities. However, automatic 3D structure prediction using current methodologies is ineffective. In this article, we provide a reliable computational method for the quick folding of RNA molecules. We create a condensed RNA model including base-pairing and base-stacking interactions for discrete molecular dynamics (DMD) simulations. We show accurate folding of 150 structurally different RNA strands. The bulk of experimental structures and DMD-predicted 3D structures differ by 4 A. Native base-pair interactions make up 94% of the secondary structures that match to the expected 3D structures. In DMD simulations, the folding thermodynamics and kinetics of tRNAPhe, pseudoknots, and mRNA fragments are consistent with earlier experimental results. Transient, non-native conformations are present during RNA molecule folding, which suggests nonhierarchical RNA folding. With linearly rising computational time with RNA length, our technique enables quick conformational sampling of RNA folding. We see this method as a useful tool for studying the structural and functional aspects of RNA [9].

DISCUSSION

Discrete dynamics, also known as discrete-time dynamical systems or difference equations, is a branch of mathematics that studies the evolution of systems over time, where the state of the system is updated at discrete intervals. Unlike continuous dynamical systems, where time is continuous, discrete dynamics deals with systems where time is modeled as a sequence of discrete steps. The study of discrete dynamics has numerous applications in many fields, including physics, biology, and economics, engineering, and computer science. In this discussion, we will explore some key concepts and applications of discrete dynamics. In discrete dynamics, a system is defined by a set of equations that describe how the state of the system evolves over time. The state of the system is represented by a set of variables, and the equations describe how these variables change from one time step to the next. The time steps are usually uniform, and the equations are typically of the form:

X_{n+1} = f(X_n)

where X_n represents the state of the system at time n, and f is a function that maps the current state of the system to its next state. One of the fundamental concepts in discrete dynamics is the fixed point, which is a state of the system that does not change over time. In other words, if X is a fixed point of the system, then $f(X) = X$. Fixed points are important because they represent the long-term behavior of the system, and they can be stable or unstable, depending on the behavior of the system in the neighborhood of the fixed point.

Another important concept in discrete dynamics is the periodic orbit, which is a sequence of states that repeat periodically over time. A periodic orbit can be of any period, and it can be stable or unstable, depending on the behavior of the system in the neighborhood of the orbit. Discrete dynamics has numerous applications in many fields. In physics, it is used to model the behavior of discrete systems, such as particles or atoms. In biology, it is used to model the growth and evolution of populations, as well as the dynamics of biochemical networks. In economics, it is used to model the behavior of financial markets, as well as the dynamics of economic systems. In engineering, it is used to model the behavior of discrete control systems, such as digital circuits or robotic systems. In computer science, it is used to model the behavior of discrete algorithms, such as sorting or searching.

One of the most famous applications of discrete dynamics is the logistic map, which is a simple model of population growth. The logistic map is defined by the equation:

$$X_{n+1} = r X_n (1 - X_n)$$

Where X_n represents the population at time n, and r is a parameter that represents the growth rate of the population. The logistic map exhibits a wide range of behaviors, including stable fixed points, periodic orbits, and chaotic behavior, depending on the value of the parameter r. Another important application of discrete dynamics is cryptography, where it is used to design and analyze cryptographic algorithms. Cryptography is the study of techniques for secure communication in the presence of adversaries. Discrete dynamics is used in cryptography to design stream ciphers, which are cryptographic algorithms that use a sequence of keys to encrypt and decrypt messages. Stream ciphers are based on the idea of generating a pseudorandom sequence of bits that are used to encrypt the message. Discrete dynamics is used to generate the pseudorandom sequence of bits, which is then combined with the message to produce the cipher text.

Discrete dynamics is a fascinating branch of mathematics that studies the evolution of systems over time, where the state of the system is updated at discrete intervals. It has numerous applications in many fields, including physics, biology, and economics, engineering, and computer science. In computer science, discrete dynamics is used to model the behavior of algorithms, such as sorting, searching, and graph traversal algorithms. Discrete dynamics is also used in the analysis of networks, such as social networks or communication networks. In Figure 1 illustrate the real-time operating system.

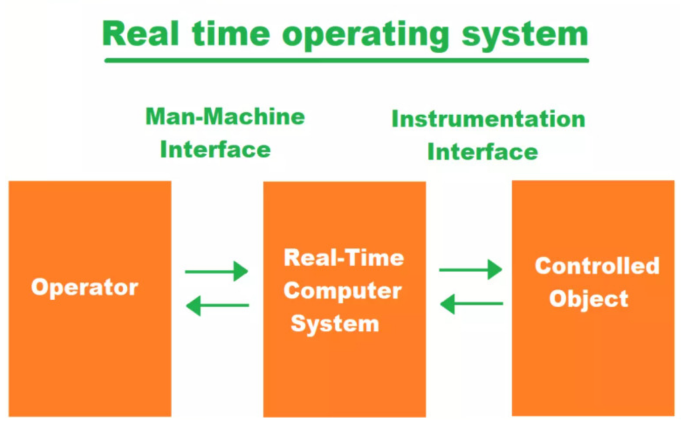

Real time operating system
Man-Machine Interface
Instrumentation Interface
Operator
Real-Time Computer System
Controlled Object

Figure 1: Illustrate the real time operating system.

One of the most important applications of discrete dynamics in computer science is the study of cellular automata. Cellular automata are discrete dynamical systems that consist of a regular grid of cells, where each cell can be in one of a finite number of states. The state of each cell is updated according to a set of local rules that depend only on the states of the neighboring cells. Cellular automata have been used to model a wide range of phenomena, including physical systems, biological systems, and social systems. They are also used in the study of complexity theory, where they are used to explore the behavior of simple systems that exhibit complex behavior.

Another important concept in discrete dynamics is chaos, which is a phenomenon that can occur in certain nonlinear dynamical systems. Chaos is characterized by a high sensitivity to initial conditions, which means that small changes in the initial conditions can lead to large differences in the behavior of the system over time. Chaos is an important concept in discrete dynamics because it can lead to unpredictable and complex behavior, which is often observed in natural systems, such as weather patterns, fluid dynamics, and population dynamics. Embedded system models might include discrete or continuous components. In general, continuous components develop gradually while discrete components do so suddenly. The chapter before this one focused on continuous components and demonstrated that the

Ordinary differential or integral equations, or equivalently actor models that reflect these equations, may often be used to simulate the physical dynamics of the system. On the other hand, ODEs do not easily represent discrete components. In this chapter, we explore the modelling of discrete dynamics using state machines. We'll demonstrate how these state machines may be linked with continuous dynamics models in the next chapter to create hybrid system models. Each arrival or departure in the aforementioned scenario is represented as a distinct event. A discrete occurrence happens at a single moment of time as opposed to gradually. The Integrator actor employed in the previous chapter is comparable to the Counter actor in Figure

The Integrator gathers input values as the Counter actor does. But it goes about it quite differently. A continuous-time signal with the function type x: R R or x: R+ R serves as the input to an integrator. On the other hand, the signal u that enters the Counter's up input port is a function of the type u: R — "absent, present." This indicates that the input u(t) is either missing at any time t R, in which case there is no event, or present at such time, in which case there is. This kind of signal is referred to as a pure signal. It has no value; all the information it conveys is simply whether it is there or not at any particular moment.

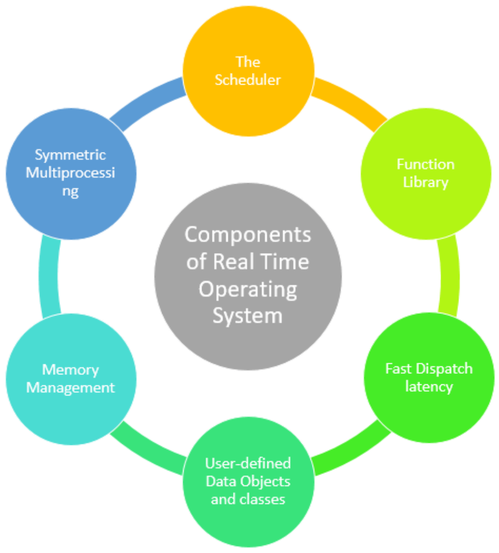

The Scheduler
Function Library
Fast Dispatch latency
User-defined Data Objects and classes
Memory Management
Symmetric Multiprocessi ng
Components of Real Time Operating System

Figure 2: Illustrate the components of operating system.

Assume that this is how our counter works. When an event is present at the up-input port, the count is increased, and the new value of the count is produced on the output. When an event is present at the down input, it reduces the count and outputs the updated count value. It doesn't provide an output at other times (when neither of the inputs is present) (the count output is absent). As a result, a function of the kind c: R missing Z may be used to simulate the signal c. Figure 1 illustrates the components of real time operating system.

Like u and d, this signal is not pure but is either present or missing. Unlike u and d, it has a value when it is present (an integer). More specifically, assume that the inputs are discrete and that they are missing most of the time. The Counter then responds sequentially to each event in the series of input events. When compared to the Integrator, which responds continually to a range of inputs, this is considerably different. A pair of discrete signals that sometimes have an event (are present) and occasionally do not are the input to the Counter (are absent). In addition, the output is a discrete signal that, when an input is present, has a value that is a natural number, and, when an input is not there, does not exist. Without input, there is obviously no need for this Counter to take any action.

It simply has to function when inputs are available. Hence, it has discrete dynamics. A system's state is logically understood to be its condition at a certain moment. In general, the system's state has an impact on how it responds to inputs. Formally, we describe the state as an encoding of anything from the past that affects how the system responds to inputs from the present or the future. The present is a recitation of the past.

State, which in this example happens to have the same value as the output at any time t, belongs to this actor. The integral of the input signal up to time t represents the actor's state at time t. We must know this value at time t in order to predict how the subsystem will respond to inputs at and after time t. There is no need for us to learn any more about the historical contributions. The present value at t completely captures their impact on the future. The initial state value, I which is required to start things off at some beginning time.

An Integrator works inside the timeline. It integrates a continuous-time input signal, producing the cumulative area under the curve produced by the input plus the beginning state as an output at each iteration. Its current condition is made up of both the starting state and that accumulated region. The Counter actor from the previous section likewise has state, which is a collection of previous input values, but it also performs discrete actions.

The Integrator's state at time t, y(t), is a real number. As a result, we declare that States = R is the Integrator's state space. States Z because the state at time t for the Counter is an integer. Assuming that a realistic parking garage contains M spots, which is a finite and nonnegative integer, the state space for the Counter actor will be defined as States = 0 to M. This is predicated on the garage not allowing more automobiles to enter than there are available slots. The Integrator's state space is unlimited (uncountably infinite, in fact). The garage counter's state space is limited. Finite-state machines are discrete models with finite state spaces (FSMs). We go on to strong analytic methods that may be used with these models.

A state machine is a representation of a discrete dynamical system that, at each reaction, maps the values of the inputs to the values of the outputs, where the map may rely on 48 Introduction to Embedded Systems, Lee & Seshia its present condition a state machine with a limited number of potential states is known as a finite-state machine (FSM). When there aren't too many states, it's easy to represent FSMs using a graphical language similar. Since each state is shown here by a bubble, the set of states for this diagram is given by States = "State1, State2, State3".

Each series of reactions starts with an initial state, State1, which is represented in the figure by a hanging arrow into it. The discrete dynamics of the state machine and the conversion of input values into output values are governed by transitions between states a transition is symbolized by a curving arrow that moves from one state to another. As State3 in the image shows, a transition might also begin and conclude at the same state. The transition in question is referred to as a self-transition.

Whether the transition can be made on a response is decided by the guard. What outputs are generated by each response are specified in the action. When the transition should be made, a guard is a predicate (a boolean-valued expression) that evaluates to true, altering the state from that at the beginning of the transition to that at the conclusion. A guard is said to be enabled when it evaluates to true. Assignment of values (or absence) to the output ports is referred to as an action. No output port undergoes a status change. The machine will stay in the same state if no guard on any transition out of the current state evaluates to true.

At a reaction, any or all of the inputs might be missing. It is still conceivable for a guard to evaluate to true in this scenario, in which case a transition is made. The machine will stall if the input is missing and no guard on any transition out of the current state evaluates to true. When both the inputs and the outputs are missing, the machine stutters and remains in the same condition. Nothing changes and no advancement is made. What would happen if the count was at 0 and a vehicle left was not stated clearly in our informal explanation of the garage counter. FSM models provide the primary benefit of defining all potential behaviours. The scenario is described by the model. The number is still at 0. As a result, formal verification may be used to test whether the given behaviours really are desired behaviors using FSM models. Such tests cannot be performed on the informal specification, at least not entirely.

Despite the appearance that the model does not specify what occurs it does so implicitly if the state is 0 and down is present; the state stays the same and no output is produced. The graphic does not specifically depict the response. Such responses may sometimes be highlighted, in which case they can be shown directly. Using a default transition, is an easy method to do this. The default transition is shown in that picture with dashed lines and is identified as "true /" If no non-default transition is active and the guard of the transition evaluates to true, the transition is enabled. Therefore, if up-down evaluates to false, the default transition is enabled, and when the default transition is taken, there is no output.

While default transitions make notation easier, they are not really required. Any default transition may be substituted by an ordinary transition with an adequately chosen guard. For instance, we may employ a standard transition with guard (up down). One technique to give precedence to transitions in a diagram is to utilise both standard transitions and default transitions. A regular transition takes precedence over a standard transition. When both have guards that are true when evaluated, the regular transition is used. More than two degrees of priority are supported by several state machine formalisms.

A particular mathematical description of the dynamics of a state machine is defined by the graphical notation for FSMs. Sometimes it's helpful to use a mathematical notation that has the same meaning as the graphical notation, especially for big state machines where the graphical notation might be cumbersome. The FSM responds in a chain of reactions. The FSM has a current state for each reaction, and that reaction may transition to a subsequent state, which will be the current state for the next reaction. These states may be numbered, with 0 being the original state. Let s: N States, specifically, be a function that outputs the state of an FSM at reaction n N. Initially, initialState = s(0).

Assign the input and output values at each response to x: N Inputs and y: N Outputs. As a result, the initial input value is represented by x(0) and the first output valuation by y(0). The equation below describes the dynamics of the state machine:

Update: (s(n + 1), y(n)

(s(n), x(n)) (3.1)

In terms of the present state and input, this provides the following state and output. All of the transitions, guards, and output requirements in an FSM are encoded by the update function. Update function is often replaced by the phrase transition function. The values of the input and output also take a naturally mathematical structure. Assume that an FSM has input ports P = p1, p2, pN, each of which has a matching type Vp.

CONCLUSION

Discrete dynamics is a vital field of mathematics that studies the evolution of systems over time in discrete intervals. It provides a powerful tool for modeling and understanding the behavior of complex systems in many fields, including physics, biology, and economics, engineering, and computer science. Discrete dynamics has numerous applications, including population growth, cryptography, cellular automata, chaos, and more. The study of discrete dynamics has led to many important discoveries and remains an active area of research today. The impact of discrete dynamics on science and technology is significant, and it will undoubtedly continue to play a crucial role in advancing our understanding of complex systems in the future.

REFERENCES:

[1] B. Bathellier, L. Ushakova, and S. Rumpel, "Discrete Neocortical Dynamics Predict Behavioral Categorization of Sounds," *Neuron*, 2012, doi: 10.1016/j.neuron.2012.07.008.

[2] S. R. Bishop, H. Momiji, R. Carretero-González, and A. Warren, "Modelling desert dune fields based on discrete dynamics," *Discret. Dyn. Nat. Soc.*, 2002, doi: 10.1080/10260220290013462.

[3] S. Toxvaerd, "Newton's discrete dynamics," *Eur. Phys. J. Plus*, 2020, doi: 10.1140/epjp/s13360-020-00271-5.

[4] Y. Gu and J. A. El-Awady, "Quantifying the effect of hydrogen on dislocation dynamics: A three-dimensional discrete dislocation dynamics framework," *J. Mech. Phys. Solids*, 2018, doi: 10.1016/j.jmps.2018.01.006.

[5] J. P. García-Sandoval, "Fractals and discrete dynamics associated to prime numbers," *Chaos, Solitons and Fractals*, 2020, doi: 10.1016/j.chaos.2020.110029.

[6] A. Cabrera, M. del Hoyo, and E. Pujals, "Discrete dynamics and differentiable stacks," *Rev. Mat. Iberoam.*, 2020, doi: 10.4171/RMI/1194.

[7] H. K. Inagaki, L. Fontolan, S. Romani, and K. Svoboda, “Discrete attractor dynamics underlies persistent activity in the frontal cortex,” *Nature*, 2019, doi: 10.1038/s41586-019-0919-7.

[8] S. S. Quek, Z. Wu, Y. W. Zhang, and D. J. Srolovitz, “Polycrystal deformation in a discrete dislocation dynamics framework,” *Acta Mater.*, 2014, doi: 10.1016/j.actamat.2014.04.063.

[9] F. Ding, S. Sharma, P. Chalasani, V. V. Demidov, N. E. Broude, and N. V. Dokholyan, “Ab initio RNA folding by discrete molecular dynamics: From structure prediction to folding mechanisms,” *RNA*, 2008, doi: 10.1261/rna.894608.

CHAPTER 4

THE HYBRID INTELLIGENT SYSTEMS

Dr. Mohammed Aarif K O, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email Id- mohammed.aarif@presidencyuniversity.in

ABSTRACT:

Hybrid systems are a type of dynamical system that combine continuous and discrete behaviors. These systems are used to model and control complex physical systems that exhibit both continuous and discrete dynamics, such as robotic systems, power systems, and transportation networks. The continuous part of the system is described by differential equations, while the discrete part is modeled as a set of logical conditions. Hybrid systems allow for the modeling of systems that switch between different modes of behavior, making them more flexible and adaptable to changing conditions.

KEYWORDS:

Control System, Discrete Behaviors, Hybrid System, Transportation, Robotics.

INTRODUCTION

Hybrid systems are a class of dynamical systems that combine continuous and discrete dynamics. The continuous dynamics describe the evolution of the system over time through differential equations, while the discrete dynamics describe the occurrence of events or changes in the system that happen at specific time instances or under certain conditions. Hybrid systems are commonly used to model and control complex physical systems that exhibit both continuous and discrete behaviors, such as robotic systems, power systems, and transportation networks. The idea of hybrid systems dates back to the early 20th century, when researchers started using differential equations to model systems that involve both continuous and discrete variables. However, the field of hybrid systems as we know it today was established in the 1980s and 1990s, with the development of new methods and tools for analyzing, designing, and verifying hybrid systems[1], [2].

Hybrid systems are typically modeled using a combination of differential equations and automata theory. The differential equations describe the continuous behavior of the system, while the automaton describes the discrete behavior. The automaton is a mathematical model that consists of a set of states, transitions, and actions. The states represent the possible configurations of the system, while the transitions represent the changes that occur when the system moves from one state to another. The actions represent the events or activities that occur when the system transitions from one state to another.

One of the main challenges in hybrid systems is dealing with the interactions between the continuous and discrete dynamics. These interactions can lead to complex behaviors that are difficult to analyze and control. For example, a system may switch between different modes of operation depending on the values of its continuous variables or the occurrence of certain events. Such switching behavior can cause the system to become unstable or exhibit unexpected behavior [3]. To address these challenges, researchers have developed a variety of methods and tools for analyzing and controlling hybrid systems. One of the key techniques is model-based control, which involves using a mathematical model of the system to design control strategies that can regulate its behavior. Model-based control is widely used in the automotive and aerospace industries to design control systems for complex physical systems.

Another important technique for hybrid systems is verification, which involves using mathematical methods to verify the correctness of the system's behavior. Verification is critical for ensuring the safety and reliability of complex physical systems, such as medical devices or transportation systems. Verification can also be used to detect and diagnose faults in the system, which can help improve its performance and reliability.

There are many different types of hybrid systems, each with its own unique characteristics and challenges. Some examples include hybrid automata, hybrid Petri nets, and hybrid dynamical systems. Hybrid automata are a popular tool for modeling and analyzing hybrid systems, as they provide a clear and intuitive representation of the system's behavior. Hybrid Petri nets are another useful tool for modeling hybrid systems, as they allow for the explicit representation of concurrency and synchronization in the system. Hybrid dynamical systems are a more general class of systems that can capture a wide range of hybrid behaviors.

Hybrid systems have many applications in industry and academia. They are widely used in the automotive and aerospace industries to design control systems for complex physical systems, such as aircraft engines or autonomous vehicles. They are also used in robotics and automation, where they are used to develop control systems for complex machines and devices. In addition, hybrid systems have applications in areas such as power systems, manufacturing, and transportation [4].

In conclusion, hybrid systems are a powerful and flexible class of dynamical systems that combine continuous and discrete behaviors. They have many applications in industry and academia, and are a critical tool for modeling and controlling complex physical systems. As the field of hybrid systems continues to grow, it is likely that we will see the development of new methods and tools for analyzing and controlling these systems, leading to further advances in fields such as robotics, automation, and

One important area of research in hybrid systems is control and optimization. Hybrid systems often require sophisticated control algorithms that can balance the trade-offs between continuous and discrete dynamics, while ensuring the system is stable and robust. Optimization techniques can also be used to design control strategies that optimize system performance or minimize certain costs, such as energy consumption or response time.

Another key area of research in hybrid systems is simulation and modeling. Hybrid systems can exhibit complex and non-linear behavior, which makes simulation and modeling critical for understanding their performance and behavior. Simulation tools can be used to predict the behavior of the system under different conditions, while modeling tools can be used to develop mathematical models that accurately capture the system's behavior [5].

Hybrid systems also have applications in the field of cyber-physical systems, which are systems that combine physical and computational components. Cyber-physical systems are becoming increasingly important in areas such as smart grids, healthcare, and manufacturing, and hybrid systems are a critical tool for modeling and controlling these systems.

Despite the many advantages of hybrid systems, there are also some challenges and limitations. One of the main challenges is scalability, as hybrid systems can become extremely complex when dealing with large-scale systems or systems with many interacting components. Another challenge is the need for efficient algorithms and tools for analysis and control, as the complexity of hybrid systems can make traditional methods impractical or infeasible.

In recent years, there has been a growing interest in the use of machine learning for hybrid systems, particularly in the area of reinforcement learning. Reinforcement learning is a type of machine learning that involves training an agent to make decisions in an uncertain and dynamic environment. Hybrid systems are a natural fit for reinforcement learning, as they often involve complex and unpredictable behaviors that can be difficult to model using traditional techniques.

Hybrid systems are a critical tool for modeling and controlling complex physical systems that exhibit both continuous and discrete behaviors. The field of hybrid systems is still evolving, and there are many exciting areas of research that are currently being explored. As the field continues to grow, it is likely that we will see the development of new methods and tools that will enable the design of more sophisticated and reliable hybrid systems for a wide range of applications.

LITERATURE REVIEW

Alexey Zhirabok et al. Investigations are made on the issue of fault identification in hybrid systems. The set of nonlinear differential equations, the finite automaton, and the so-called mode activator that coordinates the actions of these two components are supposed to make up the hybrid systems under discussion. Using sliding mode observers, the defect identification issue is solved. The recommended method for creating sliding mode observers is based on the original system's reduced order model. As a result, sliding mode observers' complexity may be reduced, and the original system's restrictions can be loosened. Examples highlight specifics of the answer.

Kiran Dhangar et al. The removal of emerging contaminants (ECs) from the wastewater matrix has been researched using a wide range of biological and physicochemical treatment techniques. Hybrid systems, which use the distinct removal capacity of the various treatment procedures, were investigated owing to the inability of these treatments to entirely degrade the ECs in wastewater. This study provides information on such hybrid systems that combine a number of physical, chemical, and biological treatments to quickly and effectively remove ECs from wastewater. In the majority of hybrid systems, biological therapies came first, followed by physical or chemical treatments. Pharmaceuticals, beta blockers, pesticides, and EDCs were all successfully removed using the hybrid system of a membrane bioreactor (MBR) and membrane filtrations (RO/NF). The biosorptive clearance of pharmaceuticals and certain beta blockers showed remarkable promise in some hybrid systems made of built wetlands and waste stabilisation ponds [6].

Yashwant Sawle et al. Due to negative environmental effects and an increase in energy prices associated with the use of traditional energy sources, renewable energy systems are anticipated to become more commonplace in the future. Renewable energy sources like solar and wind have the ability to partially address the load problem since they are complementary to one another. However, due to their unpredictable character, such answers are rarely totally reliable when independently studied. In this regard, it has been determined that autonomous photovoltaic and wind hybrid energy systems are a more economically feasible solution to meet the energy needs of multiple remote customers throughout the globe. This paper's goal is to provide an understanding of the hybrid system architecture, modelling, renewable energy sources, optimization criteria, control schemes, and software for optimum sizing. Comparison of several freestanding hybrid combinations in a distant area: a case study the PV-Wind-Battery-DG hybrid system is the most cost- and emission-efficient option out of all the other hybrid system combinations, according to Barwani, India. This report also discusses various enhancements that may be made in the near future that could increase the perceived financial appeal of these kinds of tactics and customer support for them [7].

Daniel Coles et al. performance in terms of technology, finances, and the environment of hybrid systems that combine a tidal stream or wind turbine with momentary battery storage and backup oil generators. The devices are intended to partly replace oil generators on the British Channel Islands (island of Alderney). Every day, there are four power generating times made possible by the tidal stream turbine. The oil generators can only produce 1.6 GWh per year because of the very frequent power cycling. When wind resources are low, however, the wind hybrid system must depend on the backup oil generators for extended periods of time, which results in an increase of 2.4 GWh/year (or 50%) in annual energy consumption. Because of this, the tidal hybrid system displaces more oil, saving £0.25 million per year on fuel, or £6.4 million over a 25-year working life, assuming a flat cost of oil. Tidal and wind hybrid systems successfully displace 78% and 67% of the oil, respectively the same as the reduction in carbon emissions. Two more wind turbines are required for the wind hybrid system to replace the same quantity of oil as the tidal hybrid system.

DISCUSSION

Hybrid systems are an important and rapidly growing area of research, with a wide range of applications in many different fields. One of the main advantages of hybrid systems is that they can model a wide range of complex physical systems that exhibit both continuous and discrete behaviors, including cyber-physical systems, control systems, and manufacturing systems, among others. However, despite their many advantages, there are also some challenges and limitations associated with hybrid systems. One of the main challenges is the complexity of the systems themselves, which can make it difficult to develop accurate models and effective control strategies. In addition, the need for efficient algorithms and tools for analysis and control can be a significant challenge, particularly when dealing with large-scale or highly interconnected systems.

Another challenge associated with hybrid systems is the need for interdisciplinary expertise. Because hybrid systems involve both physical and computational components, they require expertise from multiple fields, including engineering, computer science, mathematics, and physics, among others. This can make it difficult to develop integrated and cohesive approaches to modeling and controlling hybrid systems. Despite these challenges, there are many exciting areas of research in hybrid systems that are currently being explored. For example, the use of machine learning and reinforcement learning for hybrid systems is an active area of research that has the potential to significantly improve the accuracy and efficiency of control strategies for complex systems. In addition, the development of new modeling techniques and simulation tools is also an important area of research that can help to overcome some of the challenges associated with hybrid systems.

Overall, the development of hybrid systems represents an important advance in our ability to model and control complex physical systems, and has the potential to significantly improve performance and efficiency in a wide range of applications. However, as the field continues to evolve, it will be important to continue to address the challenges and limitations associated with hybrid systems, and to develop new methods and tools that enable the design of more sophisticated and reliable hybrid systems for a wide range of applications. One of the key advantages of hybrid systems is their ability to model and control complex physical systems that exhibit both continuous and discrete behavior. This is particularly important for cyber-physical systems, which are becoming increasingly important in a wide range of applications, from smart cities and transportation systems to healthcare and manufacturing.

In addition, hybrid systems can also be used to optimize system performance and minimize costs, such as energy consumption, response time, and resource utilization. This is particularly important in applications such as transportation, where minimizing energy consumption and improving efficiency can have a significant impact on both the environment and the economy.

One of the challenges associated with hybrid systems is the need for efficient algorithms and tools for analysis and control. This is particularly true when dealing with large-scale or highly interconnected systems, where traditional methods can be impractical or infeasible. However, recent advances in machine learning and reinforcement learning have the potential to significantly improve the efficiency and accuracy of control strategies for hybrid systems, and may lead to the development of new and more effective control algorithms. In Figure 1 illustrate the hybrid system.

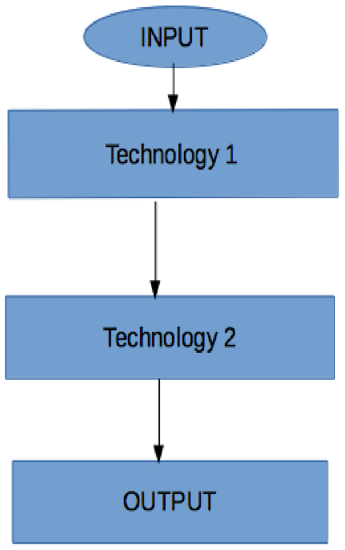


Figure 1: Illustrate the Hybrid System in Soft Computing.

Another important area of research in hybrid systems is the development of new modeling techniques and simulation tools. Hybrid systems can exhibit complex and non-linear behavior, and accurate models and simulations are critical for understanding system performance and behavior. However, traditional modeling techniques can be limited in their ability to accurately capture the behavior of complex hybrid systems, and new modeling techniques and simulation tools are needed to overcome these limitations.

Hybrid systems represent an important advance in our ability to model and control complex physical systems, and have the potential to significantly improve performance and efficiency in a wide range of applications. However, as the field continues to evolve, it will be important to continue to address the challenges and limitations associated with hybrid systems, and to develop new methods and tools that enable the design of more sophisticated and reliable hybrid systems for a wide range of applications.

Cyber-physical systems often incorporate both discrete and continuous dynamics because they merge physical dynamics with computing systems. The modelling methods may be coupled, as we demonstrate in this chapter, to produce what are referred as system hybrids. Models for hybrid systems are often simpler and easier to grasp.

Here, we demonstrate how state machines may be extended to accept continuous inputs and outputs as well as the combination of discrete and continuous dynamics. We describe how state machines contain inputs described by the set Inputs, which may be pure signals or may have a value. The state machine contains a number of input ports in both scenarios; for pure signals, these ports are either present or absent, and for valued signals, these ports have a value at each state machine response. In Figure 2 illustrate the A new hybrid model of software engineering and systems engineering for embedded system development methodology.

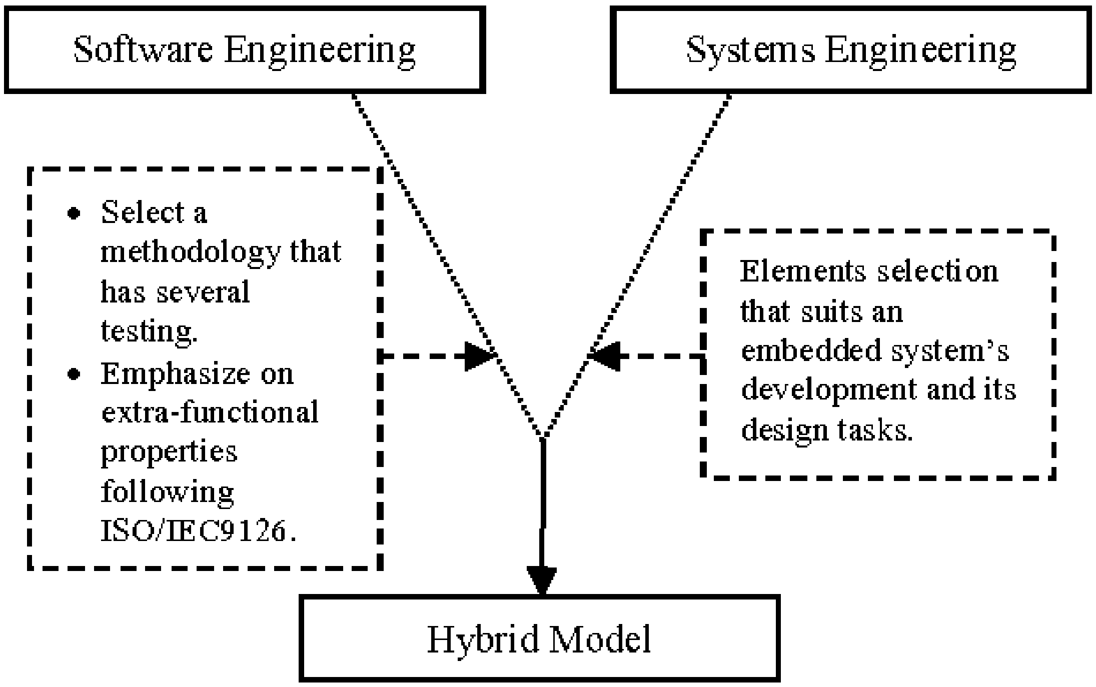

Software Engineering
Systems Engineering
• Select a methodology that has several testing.
• Emphasize on extra-functional properties following ISO/IEC9126.
Elements selection that suits an embedded system's development and its design tasks.
Hybrid Model

Figure 2: Illustrate the new hybrid model of software engineering and systems engineering.

We further clarify how actions on transitions determine the values of outputs. Ports may again convey pure signals or valued signals, and ports can also be used to represent outputs. Taking a transition in the case of a pure signal determines if the output is present or absent, while taking a transition in the case of a valued signal either assigns a value or declares that the signal is not there. It is assumed that outputs are missing in between transitions.

Given this input/output perspective on state machines, it makes sense to consider them to be actors. In that image, n input ports with the designation "i1" are what we're assuming there are. These ports have a value at each reaction that is either present or absent if the port carries a pure signal or a part of a collection of values if the port carries a valued signal. The results are comparable. The actions assign values to the output ports, while the guards on the transitions specify subsets of potential input port values.

We'll now expand this to accept continuous-time signals as inputs and outputs. State transitions must be understood to take place on the same timeline as the time-based part of the system in order for state machine models and time-based models to coexist. For this reason, the idea of discrete reactions is sufficient, but inputs and outputs between reactions are no longer necessary. Instead, a transition will be defined as taking place when a guard on an outgoing transition from the present state gets enabled. As previously, it is assumed that a state machine does not switch between modes when there are no reactions. However, it is no longer necessary for the inputs and outputs to remain missing throughout that period. In a hybrid system, the state machine's present state has a state refinement that determines the output's dynamic behavior in relation to the input. The output is constant in each stage in the previous simple example, which is a pretty easy dynamics. Hybrid systems may become quite complex.

The overall layout of a hybrid system model. A two-state finite-state machine may be seen in the illustration. There is a state refinement for each state referred to as a "time-based system" in the illustration. The outputs' dynamic behavior as well as potentially additional continuous state variables are defined by the state refinement. Additionally, each transition has the possibility to provide set actions that, when executed, set the values of these extra state variables. Because it contains a limited number of modes one for each FSM state and has dynamics that are determined by the state refinement while in a mode, a hybrid system is sometimes referred to as a modal model. As we shall see, referring to the states of the FSM as modes rather than states helps to avoid misunderstanding with the states of the refinements.

Hybrid systems may be quite complex. We first go over a very basic form known as a timed automaton in this section. Then, using more complex forms, we demonstrate how nontrivial physical dynamics and nontrivial control systems may be modelled. The majority of cyber-physical systems need timing activities and tracking the passage of time. A clock's state advances linearly over time, making it a timekeeping device with especially straightforward dynamics.

The simplest non-trivial hybrid systems are timed automata, which are presented and which allow the building of more complex systems from such basic clocks. They are modal models, and the only thing the time-based improvements do is quantify the passage of time. A first-order differential equation, t Tm, s (t) = a, is used to simulate a clock. In this equation, s: R R is a continuous-time signal, s(t) is the value of the clock at time t, and Tm R is the portion of time that the hybrid system is operating in mode m. While the system is operating in this mode, the clock's rate, a, remains constant. Exercise 10 explores an intriguing challenge in the preceding case of the bouncing ball. In particular, as time goes on, the interval between bounces grows shorter. Actually, it

Zeno systems are named after Zeno of Elea, a pre-Socratic Greek philosopher well known for his paradoxes, and refer to systems having an infinite number of discrete events occurring over a limited period of time. The ball will naturally cease bouncing in the real world; the Zeno behaviour is a result of the model. A control system consists of four parts: the system known as the plant, which represents the physical process that needs to be controlled; the environment in which the plant operates; the sensors, which measure some environmental and plant-related variables; and the controller, which chooses the time-based inputs for the plant. The supervisory control, which establishes the mode transition structure, and the low-level control, which establishes the time-based inputs to the plant, are the two levels of the controller. The supervisory controller intuitively chooses which of numerous techniques should be used, and the low-level controller carries out the decision. The best way to simulate such two-level controllers is via hybrid systems.

In particular, deep learning methods, such as neural networks and deep reinforcement learning, have shown the ability to learn complex, non-linear, and high-dimensional models of hybrid systems. These methods can also be used to develop effective control strategies that can adapt to changing system dynamics in real-time. Another promising area of research is the use of hybrid systems for the design of safe and secure cyber-physical systems. In particular, formal methods, such as model checking and runtime verification, can be used to verify the correctness and safety of hybrid systems. These methods can also be used to detect and respond to security breaches and cyber-attacks on cyber-physical systems[8].

Moreover, recent advances in hybrid systems have also led to the development of new modeling and control techniques. For example, reachability analysis, which is a technique for analyzing the set of reachable states of a hybrid system, has been used to develop efficient algorithms for safety and stability analysis. Model predictive control, which is a control strategy that uses an online optimization algorithm to optimize the system behavior, has also been used to develop effective control strategies for hybrid systems.

Hybrid systems are a powerful and versatile tool for modeling and controlling complex physical systems. They have a wide range of applications in many different fields, and their behavior can be analyzed using simulation, verification, and optimization techniques. However, there are also some challenges and limitations associated with hybrid systems, including the complexity of the systems, the need for efficient algorithms and interdisciplinary expertise, and the difficulty of predicting and controlling non-linear and complex behavior. Recent advances in machine-learning-based techniques, formal methods, and new modeling and control techniques have shown great promise in addressing these challenges and limitations and advancing the field of hybrid systems[9], [10].

One important aspect of hybrid systems that has gained increased attention in recent years is their ability to capture the dynamics of biological systems. Hybrid systems have been used to model a wide range of biological processes, including gene regulatory networks, metabolic pathways, and signal transduction pathways. These models can be used to gain insights into the behavior of biological systems, as well as to design new drugs and therapies for diseases.Another area of research that has seen significant progress in recent years is the development of hybrid electric vehicles (HEVs). HEVs are vehicles that use a combination of electric motors and internal combustion engines to power the vehicle. The complex dynamics of HEVs can be effectively modeled using hybrid systems, and optimal control strategies can be developed to maximize the fuel efficiency and reduce the emissions of these vehicles.

In addition to their applications in engineering and biology, hybrid systems also have important implications for the design and analysis of social and economic systems. Hybrid models have been used to model complex systems such as the stock market, transportation systems, and power grids. These models can be used to predict and control the behavior of these systems, as well as to design more efficient and sustainable systems.

Finally, it is worth noting that hybrid systems are not just a theoretical concept. There are many real-world applications of hybrid systems, ranging from aerospace and robotics to energy and transportation. In fact, many of the technologies and systems that we use every day, such as smartphones, electric cars, and power grids, rely on hybrid systems to function effectively. The recent advances in machine learning, formal methods, and new modeling and control techniques have helped to address some of the challenges and limitations of hybrid systems, and have opened up new opportunities for research and innovation. As such, hybrid systems will undoubtedly continue to play an important role in the development of new technologies and the understanding of complex systems for years to come.

CONCLUSION

Hybrid systems have great potential to revolutionize the way we understand and design complex systems, and will undoubtedly play an increasingly important role in the development of new technologies and the analysis of complex systems in the future. By combining different modeling and control techniques, hybrid systems offer a powerful framework for addressing complex, non-linear, and high-dimensional systems, and have already made significant contributions to a wide range of fields. With continued research and development, hybrid systems are likely to continue to push the boundaries of what is possible in modeling and controlling complex systems, and unlock new opportunities for innovation and progress.

REFERENCES:

[1] A. Jain, O. Prakash, A. Talukdar, S. P. Samal, R. Nanjundappa, and N. Mahesha, “Screen intelligence engine: ML based method for predicting IoT devices using screen contents,” in *2019 IEEE 16th India Council International Conference, INDICON 2019 - Symposium Proceedings*, 2019. doi: 10.1109/INDICON47234.2019.9030306.

[2] L. A. Dobrzański and A. D. Dobrzańska-Danikiewicz, “Why are carbon-based materials important in civilization progress and especially in the industry 4.0 stage of the industrial revolution,” *Materials Performance and Characterization*. 2019. doi: 10.1520/MPC20190145.

[3] S. C. Neves, L. Moroni, C. C. Barrias, and P. L. Granja, “Leveling Up Hydrogels: Hybrid Systems in Tissue Engineering,” *Trends in Biotechnology*. 2020. doi: 10.1016/j.tibtech.2019.09.004.

[4] X. Ju, C. Xu, Y. Hu, X. Han, G. Wei, and X. Du, “A review on the development of photovoltaic/concentrated solar power (PV-CSP) hybrid systems,” *Sol. Energy Mater. Sol. Cells*, 2017, doi: 10.1016/j.solmat.2016.12.004.

[5] T. Yan *et al.*, “A critical review on membrane hybrid system for nutrient recovery from wastewater,” *Chemical Engineering Journal*. 2018. doi: 10.1016/j.cej.2018.04.166.

[6] K. Dhangar and M. Kumar, “Tricks and tracks in removal of emerging contaminants from the wastewater through hybrid treatment systems: A review,” *Science of the Total Environment*. 2020. doi: 10.1016/j.scitotenv.2020.140320.

[7] Y. Sawle, S. C. Gupta, and A. K. Bohre, “PV-wind hybrid system: A review with case study,” *Cogent Engineering*. 2016. doi: 10.1080/23311916.2016.1189305.

[8] Syaifuddin, P. T. Nguyen, Q. L. H. ThuyTo Nguyen, V. D. B. Huynh, and K. Shankar, “Substantial interest of business intelligence: An implementation approach for used in practice,” *Test Eng. Manag.*, 2019.

[9] S. Eom, R. M. Voyles, and D. Kusuma, “Embedding intelligence into smart tupperware brings internet of things home,” in *Annual Technical Conference - ANTEC, Conference Proceedings*, 2019.

[10] E. Bouzekri *et al.*, “Engineering issues related to the development of a recommender system in a critical context: Application to interactive cockpits,” *Int. J. Hum. Comput. Stud.*, 2019, doi: 10.1016/j.ijhcs.2018.05.001.

CHAPTER 5
COMPOSITION OF STATE MACHINES

Mrs. Kehkeshan Jalall S, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email Id- kehkeshan@presidencyuniversity.in

ABSTRACT:

The composition of state machines is a powerful technique for modeling complex systems. State machines are a widely used formalism for modeling the behavior of systems that change over time. In many cases, complex systems can be decomposed into smaller, more manageable subsystems, each of which can be modeled using a state machine. The composition of these state machines can then be used to model the behavior of the entire system. This approach allows for modular and scalable modeling, and can help to reduce the complexity of the overall system. This paper provides an overview of the composition of state machines, including the different techniques for composition, the advantages and limitations of the approach, and some of the applications of the technique in various fields.

KEYWORDS:

Complex System, Composition, Powerful Technique, Scalable Modelling, Machines.

INTRODUCTION

The composition of state machines is a fundamental concept in the field of formal methods for modeling and analyzing complex systems. State machines are widely used to model the behavior of systems that change over time, such as control systems, communication protocols, and software applications. In many cases, these systems can be decomposed into smaller, more manageable subsystems, each of which can be modeled using a state machine. The composition of these state machines can then be used to model the behavior of the entire system [1].

The composition of state machines allows for modular and scalable modeling, where each subsystem can be developed and analyzed independently, and the overall system behavior can be analyzed by combining the behaviors of the individual subsystems. This approach helps to reduce the complexity of the overall system, and allows for the development of complex systems in a more manageable and efficient manner. In recent years, the composition of state machines has gained increased attention in the field of formal methods, as well as in related fields such as software engineering and control theory. This is due in part to the increasing complexity of modern systems, which require more sophisticated modeling and analysis techniques, as well as the availability of new tools and techniques for modeling and analyzing state machines.

In this paper, we provide an overview of the composition of state machines, including the different techniques for composition, the advantages and limitations of the approach, and some of the applications of the technique in various fields. We also discuss some of the challenges and open research questions in the area of state machine composition, and highlight some of the current and future directions of research in this area [2].

The composition of state machines is a powerful technique for modeling and analyzing complex systems. It allows for the decomposition of a system into smaller, more manageable subsystems, which can be modeled using individual state machines. These state machines can then be composed to model the behavior of the entire system. In this section, we will explore the different techniques for composing state machines, the advantages and limitations of the approach, and some of the applications of the technique in various fields.

Techniques for Composing State Machines

There are several techniques for composing state machines, each of which has its advantages and limitations. The most commonly used techniques for composing state machines include:

1. Parallel Composition: In parallel composition, the state machines are executed in parallel. This technique is used when the subsystems are completely independent of each other and do not interact with each other.

2. Sequential Composition: In sequential composition, the state machines are executed sequentially, where the output of one subsystem is used as the input of the next subsystem. This technique is used when the subsystems interact with each other and depend on each other's outputs.

3. Hierarchical Composition: In hierarchical composition, the state machines are organized into a hierarchical structure, where the top-level state machine controls the behavior of the lower-level state machines. This technique is used when the subsystems can be divided into smaller subsystems, and the top-level state machine controls the behavior of the lower-level state machines.

4. Feedback Composition: In feedback composition, the output of a subsystem is fed back to its input, which can create loops in the state machine. This technique is used when the subsystems interact with each other and depend on each other's outputs.

The composition of state machines offers several advantages over other modeling and analysis techniques. One of the primary advantages is modularity. By decomposing a system into smaller subsystems, each of which can be modeled using an individual state machine, the overall complexity of the system can be reduced. This allows for more efficient modeling and analysis, as well as easier maintenance and modification of the system. State machine composition also offers scalability. As the size and complexity of a system increase, it becomes more difficult to model and analyze the system as a whole. By decomposing the system into smaller subsystems, each of which can be modeled using an individual state machine, the overall system can be scaled to meet the needs of the application[3]–[6].

Another advantage of state machine composition is the ability to handle non-deterministic behavior. In complex systems, it is often impossible to predict the behavior of the system with certainty. State machine composition allows for the modeling of non-deterministic behavior by allowing for multiple possible outcomes for each state. Despite these advantages, state machine composition also has some limitations. One of the primary limitations is the need for well-defined interfaces between subsystems. If the interfaces are not well-defined, the behavior of the overall system may not be predictable or controllable. This requires careful design and analysis of the interfaces between subsystems.

Another limitation of state machine composition is the complexity of the composition process. As the number of subsystems and the complexity of the system increase, the process of composing the state machines can become quite complex and time-consuming. This requires specialized expertise in formal methods and modeling and analysis techniques [7].

The composition of state machines has numerous applications in various fields. In software engineering, state machine composition is used to model the behavior of software applications, and to develop software architectures that are modular and scalable. State machine composition is also used in the design of control systems, such as robotics and automation, where complex systems can be decomposed into smaller subsystems and controlled using state machines.

LITERATURE REVIEW

Ruowei Xiao et al. An unprecedented possibility for a broad range of applications is provided by the developing area of the Internet of Things (IoT). The majority of apps, however, have been using closed technology stacks and proprietary ways to integrate IoT devices. High customization costs and constrained component reusability are caused by the monolithic, mostly vendor-specific development architecture. Full-fledged IoT applications are hampered in cross-organizational, all-purpose, and dynamically changing circumstances. With the help of this study, IoT fast prototyping will be made interoperable, affordable, and user-customizable. Each IoT component, whether it be a physical device or control logic, is abstracted into a separate web service under this architecture, which is characterized by a collection of transferable states. IoT components are further put together into adaptable apps by concatenating a legitimate chain of state transfers across web services. This study establishes a Finite-State-Machine (FSM) model-driven architecture and offers the Hyper Sensor Markup Language (HSML) as an example implementation of the suggested architecture. Two real-world use cases and associated assessments are also covered [8].

D. Q. Zhao et al. A high-entropy alloy is a major component-free alloy design idea. This idea implies that there will be more intermediate metastable states when the high-entropy alloy transitions from a high-energy state to a low-energy one, in addition to referring to the complexity of alloy compositions. Changes in the microstructure's degree and kind of order correspond to distinct states. In this work, we combined elemental features with long-term ordering using machine learning, and we produced 87% prediction accuracy. This data-driven approach may speed up the identification of prospective compositions by correlating elemental properties with metastable states.

Harald Fecher et al. a compositional operational semantics for the composition of state machines in the UML. Each state machine explains how a class of item behaves. A new activity group, which is a single-threaded collection of objects, is formed if a class of a newly generated object is active. State machine communication between activity groups is different from that inside an activity group. We offer two parallel combinators that reflect this distinction and that, if their arguments are SOS, return SOSs and SOS for each state machine taken separately. Elsevier B.V. has all rights reserved as of 2006.

Shufeng Kong et al. Machine learning has quickly changed the way that materials science predicts their properties. The limited development of techniques to learn the underlying interactions of multiple elements as well as the relationships among multiple properties to facilitate property prediction in new composition spaces is one obstacle preventing full capitalization of recent advancements in machine learning. In order to solve these problems, we present the Hierarchical Correlation Learning for Multi-property Prediction (H-CLMP) framework, which seamlessly combines the following three methods: I prediction using just the composition of a material; (ii) learning and exploitation of correlations among target properties in multi-target regression; and (iii) leveraging training data from tangential domains via generative transfer learning. It is shown how the model predicts the spectrum optical absorption of 69 different three-cation metal oxide composition spaces for complicated metal oxides. H-CLMP extends the scope of machine learning to the identification of materials with extraordinary characteristics by effectively predicting non-linear composition-property connections in composition spaces for which no training data are available. This accomplishment is the product of the carefully considered combination of attention models, generative transfer learning, latent embedding learning, and property correlation learning.

Priya Sundar et al. The relevance of security offered by Web Services during Web Service Composition is highlighted by the revolution caused by Web Services as a solution to business and corporate application integration. The dynamic and erratic character of the Web makes it a difficult undertaking to satisfy the security standards. In order to develop high level business processes that perfectly fit and adhere to the demands of the service requestor, web services are combined in a process called web service composition. It entails identifying, absorbing, and delivering basic services in order to customize services often. A policy-based framework for giving security throughout the web service creation process is suggested in our article. Policies designed for efficient and secure composition examine and confirm the circumstances under which a web service job is accepted or denied. Rules are represented by nodes in the finite state machine. Transition between rules is prompted by the successful execution of policies that are specified in the node access points. When integrating the policies of elementary services, a service composition is only considered effective if there are no instances of policy breaches. Security for the composite web service is provided by the simulated FSM, which extracts the rules and policies of the web services and accurately matches and meets the policy restrictions set in the access points [9].

Christopher J. Bartel et al. It has been said that machine-learned models for the formation energy of compounds may approach the accuracy of Density Functional Theory, which is a revolutionary tool for the effective prediction of material characteristics (DFT). Five newly published compositional models, a baseline model based only on stoichiometry, and a structural model are among the models investigated in this study. Using the Materials Project database of DFT calculations for 85,014 different chemical compositions, we tested seven machine learning models for formation energy on stability predictions. Our results demonstrate that while formation energies can be accurately predicted, all compositional models perform poorly at predicting the stability of compounds, making them much less useful than DFT for the discovery and design of new solids. Most importantly, only the structural model is capable of effectively determining whether materials are stable in sparse chemical environments where few stoichiometry's yield stable molecules. With the restriction that for every new composition, the ground-state structure is not known a priori, structural models are encouraged to be used for materials discovery due to their notable no incremental improvement over compositional models. This study shows that correct formation energy forecasts do not always imply accurate stability predictions, highlighting the significance of evaluating model performance on stability predictions, for which we propose a set of publically accessible tests [10].

William Fitzhugh et al. In the realm of energy storage, solid-state batteries (SSBs) stand out as one of the most promising developments. However, the complicated electro-chemical processes that necessarily take place at the interface of solid-state electrolyte (SSE) particles are now limiting the development of SSBs. Moreover, there is no simple approach for dealing with these interface instabilities because of the material complexity of such systems. In this work, we analyse and create solid-state solid-electrolyte-interphases (SEI) with controllable electrochemical stabilities utilising a combined high-throughput ab initio computing and machine learning technique. Machine learning indicates that the mechanical constriction effect's capacity to stabilise a solid-state SEI is a nonconvex and nonlinear, but nonetheless predictable, function of composition. The intersection of the glass and ceramic sulphide families of solid electrolytes serves as a demonstration of the effectiveness of this strategy.

DISCUSSION

One of the challenges in state machine composition is the identification of appropriate interfaces between subsystems. The interfaces need to be well-defined to ensure that the composed state machine is correct and behaves as expected. In addition, the composition process can be complex, especially when dealing with non-deterministic behavior and interactions between subsystems. Techniques such as model checking and simulation are often used to verify the correctness of the composed state machine and identify any potential issues. Another challenge in state machine composition is scalability. As the number of subsystems and the complexity of the system increases, the composition process becomes more challenging. This can lead to issues such as state space explosion, where the number of states and transitions in the composed state machine becomes too large to manage. Various techniques have been developed to address this issue, such as abstraction and hierarchical composition.

State machine composition has numerous applications in various fields, including software engineering, control systems, biological systems, and communication protocols. In software engineering, state machine composition is used to model complex software systems and ensure their correctness. In control systems, it is used to model and analyze complex systems, such as power grids and autonomous vehicles. In biological systems, it is used to model complex biological processes and interactions between multiple subsystems. In communication protocols, it is used to model the behavior of communication protocols and ensure their correctness state machine composition is a powerful technique for modeling and analyzing complex systems. It allows for the decomposition of a system into smaller, more manageable subsystems, which can be modeled using individual state machines. These state machines can then be composed to model the behavior of the entire system. State machine composition has numerous applications in various fields and offers several advantages, such as modularity, scalability, and the ability to handle non-deterministic behavior. However, it also has some limitations and challenges, such as the need for well-defined interfaces between subsystems and the complexity of the composition process. As the complexity of systems continues to increase, the composition of state machines will become increasingly important for modeling and analyzing complex systems.

To further illustrate the importance and application of state machine composition, let's consider a specific example of a complex system - an automated manufacturing assembly line. This system consists of several subsystems, including a conveyor belt, robotic arms, sensors, and controllers. Each of these subsystems has its unique behavior and interactions with other subsystems. For example, the conveyor belt moves the workpieces from one workstation to another, while the robotic arms assemble the parts and the sensors detect any defects in the products. The controllers ensure that the entire system behaves as expected and coordinates the interactions between the subsystems.

To model the behavior of the automated manufacturing assembly line, state machine composition can be used. Each subsystem can be modeled using an individual state machine, capturing its unique behavior and interactions with other subsystems. For example, the state machine for the conveyor belt can have states such as "moving forward," "stopping," and "reversing," and transitions triggered by the signals from the sensors or the controllers. The state machine for the robotic arms can have states such as "picking up parts," "assembling," and "releasing parts," and transitions triggered by the signals from the sensors or the controllers.

Once the individual state machines are modeled, they can be composed to form the behavior of the entire automated manufacturing assembly line. The composed state machine captures the interactions between the subsystems and ensures that the entire system behaves as expected. The composed state machine can be analyzed using techniques such as simulation and model checking to ensure that it is correct and behaves as expected. In Figure 1 illustrate the structure of a state machine.

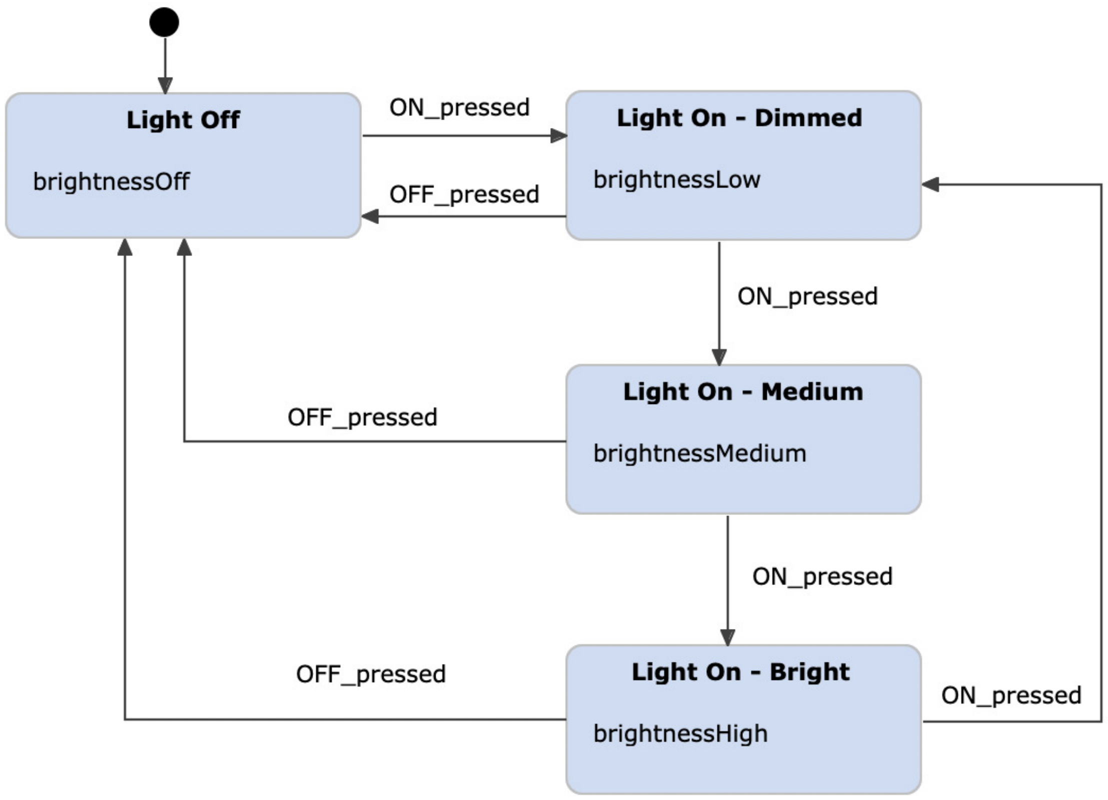


Figure 1: Illustrate the state machine.

State machine composition also offers modularity, which means that individual subsystems can be replaced or modified without affecting the behavior of the entire system. For example, if a new sensor is added to the automated manufacturing assembly line, the state machine for the sensor can be modified, and the composed state machine can be updated accordingly without affecting the behavior of other subsystems state machine composition is a powerful technique for modeling and analyzing complex systems. It allows for the decomposition of a system into smaller, more manageable subsystems, which can be modeled using individual state machines. These state machines can then be composed to model the behavior of the entire system. State machine composition has numerous applications in various fields, and it offers several advantages, such as modularity, scalability, and the ability to handle non-deterministic behavior.

State machines provide a practical method for simulating system behavior. The fact that most fascinating systems have a very large and sometimes infinite number of states is one drawback for them. Huge state spaces can be handled by automated methods, but humans struggle more with any direct depiction of a large state space. Engineering has long held the belief that complex systems should be seen as combinations of smaller systems. State machines are used in this chapter to accomplish a variety of tasks. However, the reader should be aware that there are a variety of subtle variations in how state machines might be composed. Things that on the surface seem to be same may imply various things to different individuals. A model's notational guidelines are referred to as its syntax, while the semantics of the notation are referred to as its meaning.

A number or expression is placed before the addition symbol (+) and another is placed after it in the conventional arithmetic syntax. As a result, whereas 1+ is an invalid arithmetic expression, 1+2, which consists of three symbols, is. The two integers are added when the phrase 1 + 2 is used. "The number three, acquired by adding 1 and 2," is what this statement indicates. Although the expression 2 + 1 has a different syntax, its semantics are the same (because addition is commutative).

The majority of the models in this book employ a visual syntax, where the components are boxes, circles, arrows, etc. rather than characters from a character set, and where the placement of the elements is not need to be in a certain order. These syntaxes lack the same level of standardization as, say, the syntax of mathematics. We'll see that the same syntax may have several meaning variations, which can be quite confusing. Harel invented State charts, a now-familiar language for concurrent construction of state machines (1987). Although they are all based on the same basic work, State charts have undergone significant changes (von der Beeck, 1994). These variations often give the same syntax a distinct semantic meaning.

We use the assumption that extended state machines use an actor model with the syntax describes the semantics of one such state machine in detail. The semantics that may be applied to compositions of many such machines will be covered in this chapter. Concurrent composition is the first compositional method we take into account. At least two state machines respond concurrently or separately. The process is known as synchronous composition if the reactions occur simultaneously. If they are autonomous, we refer to it as 110 Introduction to Embedded Systems, Lee & Seshia. State machine composition that is asynchronous. However, there are very small semantic changes even within these forms of composition. These changes largely concern the state machines' ability to exchange variables and interact with one another.

Hierarchy is the second compositional method we'll look at. Complex systems may also be seen as composites of smaller systems thanks to hierarchical state machines. We'll see once again that there are nuances in semantics that may exist.

We will work our way through a series of pattern compositions in order to investigate concurrent composition of state machines. Systems of any complexity may be created by combining these patterns. With side-by-side composition, when the state machines being combined do not communicate, we start with the simplest scenario. We next take into account permitting communication via shared variables, demonstrating how this generates important complexities that might make modelling more challenging. Then, we think about communication over ports, starting with arbitrary linkages and moving on to serial composition. For each sort of composition, we take into account both synchronous and asynchronous composition.

Synchronous refers to anything that is (1) present at the same moment as another thing or (2) moving or working at the same speed. The phrase has a multitude of interpretations in engineering and computer science that are largely in line with these definitions, but strangely at odds with one another. Synchronous communication is a rendezvous form of communication where the sender of a message must wait for the receiver to be ready to receive and the receiver must wait for the sender. Synchronous communication is used to describe concurrent software built utilizing threads or processes. Conceptually, the two threads see the communication as happening simultaneously, in line with the definition (1). The Java term synchronized designates code blocks that are not allowed to run concurrently. Oddly, two synchronized code blocks cannot "occur" (run) simultaneously, which contradicts both definitions.

The term "synchronous" has a third meaning in the context of software, and in this chapter, we will utilize this third definition. The foundation of the synchronous languages is this third meaning. These languages are based on two main principles. To start, the outputs of programmed components are theoreticallysimultaneous with their inputs this is called the synchrony hypothesis. Second, a program's components theoretically operate concurrently and instantly. The outputs and inputs are neither really simultaneous, nor do genuine executions happen instantly, but a good execution must act as if they were. Both of the aforementioned definitions are congruent with the way the term "synchronous" is used in this context; component executions take place simultaneously and move at the same speed.

The term "synchronous" in circuit design describes a design approach in which latches are driven by clocks that are dispersed across a circuit and record their inputs on edges of the clock. Circuits between latches must have enough time to settle in between clock edges. This approach and the model used in synchronous languages are conceptually comparable. The synchronization hypothesis is analogous to assuming that there is no delay in the circuits between latches, and global clock distribution results in simultaneous and instantaneous execution.

Synchronous refers to electrical waveforms that have the same frequency and phase in power systems engineering. Synchronous in signal processing refers to signals that have the same sample rate or sample rates that are constant multiples of one another. This second definition of the word synchronous is the foundation of the concept of synchronous dataflow. In Figure 2 shows the Finite state machine.

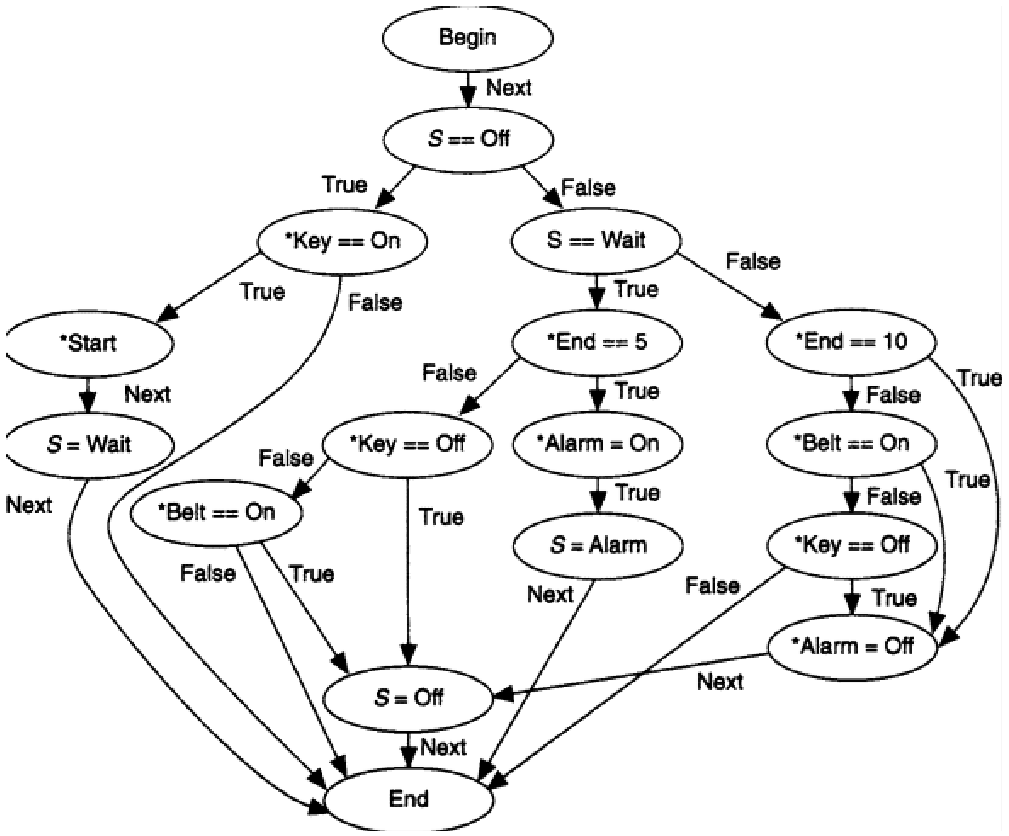


Figure 2: Illustrate the Finite State Machine.

The first compositional pattern we'll look at side-by-side composition. In this design, it is assumed that the two actors' inputs and outputs are not connected, meaning that the state machines are not in communication. Actor A has input i1 and output o1 in the illustration, whereas Actor B has input i2 and output o2. With inputs i1 and i2 and outputs o1 and o2, the two actors' combination is itself an actor C.

If the two actors are just extended state machines containing variables, then those variables will likewise be disjoint in the simplest case. We'll talk about what occurs when the variables between the two state machines are shared later. A reaction of C is an instantaneous reaction of A and B under synchronous composition. Both A and B have a single pure output, designated as a. Two pure outputs, a and b, are present in the side-by-side composition C. If the composition is synchronous, then a will be missing on the initial reaction while b will be there. It will be the opposite for the second response.

For various reasons, synchronous side-by-side composition is straightforward. First, keep in mind that the environment controls when a state machine responds. The environment need not be aware that C is a composite of two state machines in synchronous side-by-side composition. Such compositions are modular in the sense that the composition transforms into a part that may be added to other compositions as if it were an individual atomic part.

Furthermore, the synchronous side-by-side composition is deterministic if the two state machines A and B are. If a property possessed by a component is also a property of the composition, we say that property is compositional. Determinism is a feature of composition for synchronous side-by-side composition.

In addition, an FSM is itself a synchronous side-by-side composition of finite state machines. Creating a single state machine specifically for the composition is a rigorous technique to specify its semantics. Assume state machines A and B are the same. Keep in mind that the inputs A and B are two sets of values. Each valuation in the collection consists of the values that are assigned to ports. Inputs = Inputs A Inputs B means that a valuation of the inputs into C must take into account both the valuations for the inputs into A and the values for the inputs into B.

As is customary, the single FSM C might be expressed visually rather than symbolically, as seen in the example that follows a single state machine that explains the semantics of the state machines. The individual state machines in an asynchronous composition of state machines respond individually. This sentence is ambiguous and may be taken in a number of different ways. Each interpretation provides the composite a semantics. How to define a response of the composition C

a) Semantics 1. When the decision is nondeterministic, a response of C is a reaction of either A or B.
b) Semantics 2. When the decision is nondeterministic, a response of C is a reaction of A, B, or both A and B. This possibility's variation may let neither to respond.

A or B never respond simultaneously in semantics, which is known as an interleaving semantics. Their responses follow each other in a certain sequence. The fact that these semantics machines A and B may entirely overlook input events is a key detail. In a reaction where B reacts instead of A due to a nondeterministic decision, an input to C intended for machine A may be present. If this is unfavorable, schedule flexibility or synchronous composition become preferable options.

State machine composition of the state machines that provides the semantics of asynchronous side-by-side composition produces the composition state machine. It is nondeterministic in this machine. When C responds, it may move from state (s1, s3) to (s2, s3) and emit no output or it can go to (s1, s4) and emit b. Keep in mind that if we had selected Semantics 2, it would also have been possible to proceed to (s2, s4).

The definitions of States C, Inputs C, Outputs C, and initialState C for asynchronous composition under semantics 1 are the same as those for synchronous composition and are provided in (5.1) through (5.4). (5.4). The update function, however, is different, becoming updateC((sA, sB),(iA, iB)) = ((s 0 A, s0 B),(o 0 A, o0 B)), where either (s 0 A, o0 A) = updateA(sA, iA) and s 0 B = sB and o 0 B = absent or (s 0 B, o0 B) = updateB().

The decision of which component machine responds in the situation of semantics 1 and 2 presented is nondeterministic. The model doesn't explicitly state any restrictions. Introduce some scheduling strategies where the environment may affect or regulate the nondeterministic decision; this is often more beneficial. As a result, the following two extra semantics for asynchronous composition are possible:

a) Semantics 3. A response of C is a reaction of one of A or B, where which of A or B reacts is determined by the environment.

b) Semantics 4. A response of C may be caused by either reaction A, B, or both A and B, depending on the environment.

A composition must provide some kind of method for the environment to choose which component machine should respond (for scheduling the component machines) in order to achieve semantics 3 and 4. This indicates that the hierarchy is not entirely effective. Actor C must reveal more of its internal workings than merely its ports and response capabilities.

In another sense, because determinism is not maintained by composition, semantics 1 and 2 are less compositional than semantics 3 and 4. Deterministic state machines are distinct from combinations of deterministic state machines. Observe also that any behavior under semantics 3 is also a behavior under semantics 1, demonstrating that semantics is an abstraction of semantics 3. Asynchronous composition is dangerous because of the minute variations among these options. To ensure that the semantics being employed are explicit, much attention must be used. Local variables in an extended state machine may be read and written as a component of making a transition. Allowing these variables to be shared across many machines while creating state machines might be advantageous at times. Such shared variables may be particularly helpful for simulating interrupts and threads, which were covered. To make sure that the semantics of the model match those of the programmed that uses interrupts or threads, nevertheless, much attention must be taken. The memory consistency model and the idea of atomic operations are only two of the many complexities that appear.

The servers share a queue of requests since it takes an unpredictable length of time to process each request. Even though a request comes in via the first server's network interface, if one server is busy, the other server may still handle it. Assuming asynchronous composition under semantics 1, one of the two servers is nondeterministic ally selected to respond when a request is received by the composite machine C. If the server isn't in use, the request is then fulfilled. One of two things can happen if the server is serving another request: either it will coincidentally complete serving the one it is currently serving, issuing the output done, and then move on to serving the new one, or it will increase the number of pending requests while continuing to serve the current request. The decision amongst these is nondeterministic to mimic the reality that it is impossible to predict how long it will take to fulfil a request.

If server C responds when no request is made, then server A or B will once again be randomly chosen to respond. When there are one or more pending requests and the server that replies is idle, the server switches to serving and decreases the variable pending. One of three things may occur if the server that responds is not idle. It may choose to keep completing the current request, in which case it simply returns to serving after the self-transition. Alternatively, it may complete serving the request, in which case it would go to idle if there were no pending requests, or it would move back to serving and decrement pending if there were.

The preceding example's model demonstrates the complexities of concurrent systems. First, accesses to the shared variable are atomic operations because of the interleaving semantics, which is rather difficult to enforce in reality. Second, the selection of semantics 1 in this situation makes sense since the input is sent to both component machines, therefore no input event will be lost regardless of how either component machine responds. If the two computers had separate inputs, however, this semantics would not function since requests may be overlooked. That can be avoided with the aid of semantics 2, but how should the environment go about deciding? Introduction to Embedded Systems, Lee & Seshia.

Which machine responds in the composition of state machines? What if C's response to both of the independent inputs' requests is present? What does it imply when both computers update the shared variable at the same time if we pick semantics 4 in the sidebar on page 118 to enable both machines to respond simultaneously? With an interleaving semantics, the updates are no longer atomic.

Also take note that semantic 1's asynchronous composition option permits actions that do not effectively use idle computers. Consider in particular a scenario in which machine A is fulfilling customer requests while machine B is idle. Machine A will just increase pending if the nondeterministic decision causes it to respond. The idle machine is not used until B responds as a consequence of the nondeterministic decision. In actuality, semantics 1 permits actions that never make use of a machine.

In synchronous compositions, shared variables may also be employed, although complex nuances once more surface. What should happen, in instance, if two machines write to the same shared variable at the same time that one machine reads it to evaluate a guard? Do we insist on writing before reading? What happens if the transition reading the shared variable in its guard expression also writes to the same variable? One option is to use a no deterministically selected synchronous interleaving semantics in which the component machines respond in any order. The drawback of this approach is that a combination of two deterministic machines may not be deterministic. The component machines respond in a defined sequence set by the environment or by another mechanism, such as priority, in a different implementation of the synchronous interleaving semantics.

Shared variable challenges, especially with asynchronous composition, are a reflection of the intrinsic difficulty of concurrency models with shared variables. Clean solutions need a more complex semantics, which will be covered in Chapter 6. We shall describe the synchronous-reactive model of computing in that chapter. This model provides a synchronous composition semantics that is largely compositional. We have so far thought of machine compositions without direct communication. The effects of one machine's outputs becoming another's inputs are the subject of our following discussion.

Think about the two state machines A and B that are shown in Figure 5.7. Machine A's output feeds machine B's input. Cascade composition or serial composition are two names for this kind of music. The output port o1 from A in the diagram feeds events to the input port i2 of B. Assume that o1 has the data type V1 and that i2 has the data type V2 such that o1 may either accept values from V1 or not. V1 must then equal V2 in order for this composition to be considered legitimate. According to this, every output generated by an on port o1 is a valid input for B on port i2. The composition type is validated. We need to provide some mechanism for buffering the data transmitted from A to B if we want the construction of a cascade to be asynchronous. We postpone treatment of such asynchronous composition until Chapter 6, when it will be provided by dataflow and process network models of computing. We will solely discuss synchronous composition for cascade systems.

A reaction of C in the cascade structure synchronous composition is made up of reactions of both A and B, where A reacts first, producing any output that may be present, and then B reacts. The two responses are, in a sense, simultaneous and instantaneous since, logically, we see this as happening in zero time. However, they are causally connected in that the actions of A may influence those of B.

Determine which transitions are taken under what circumstances before constructing the composition machine by first creating the state space as the cross product of the state spaces of the component machines. Even though one logically results in the other, it is crucial to keep in mind that the transitions happen simultaneously.

Assume that we want to construct this using a pedestrian crossing light model similar to the input sigR of the pedestrian light may be derived from the output sigR of the traffic signal. The meaning of the composite under synchronous cascade composition. Keep in mind that risky situations, such (green, green), which is the situation where both traffic and pedestrians have a green light, are not attainable states and are thus not shown. Machine compositions that are more complicated may be built using the fundamental building blocks of side-by-side and cascade composition. Think about, for instance, the arrangement in A1 and A3 together constitute the machine B. They are a side-by-side composition. B feeds events to A2 as part of a cascade composition with B. A2 feeds events to B, while B and A2 are also a cascade composition in the reverse direction. Cycles such as Lee and Seshia's Introduction to Embedded Systems.

Combinations of side-by-side and cascade compositions are used in arbitrary interconnections of state machines, which may result in cycles like the one seen above. Feedbacks like these pose the dilemma of whether machine B or machine A2 should respond first. In the next chapter, when we discuss the synchronous-reactive paradigm of computing, this paradox will be clarified. Hierarchical FSMs, which predate Statecharts (Harel, 1987), are discussed in this section. State charts come in a variety of forms, and there are often little semantic changes between them (von der Beeck, 1994). Here, we will choose one specific semantic version and concentrate just on some of the more straightforward elements.

State refinement is the central concept of hierarchical state machines. State B has a refinement that is another FSM with two states, C and D. The machine being in state B indicates that it is either in state C or state D. The device enters state A during startup. When the guard g2 evaluates to true, the machine moves to state B, which signifies a move to state C, the refinement's starting state. When the machine makes this transition to C, it does action a2, which (depending on whether this is an extended state machine) may result in an output event or set a variable.

Then, there are two routes out of C. Guard g1 must evaluate to true for the machine to leave B and return to A. If guard g4 must also evaluate to true for the machine to leave B and go to D. What happens if both guards g1 and g4 evaluate to true is a tricky issue. Different Statechart variations may choose differently at this moment. The machine should logically end up in state A, but which of the actions a4 or a1 or both should be executed? Such nuanced inquiries contribute to explaining why there are so many diverse Statecharts variations. We choose a certain semantics because it has appealing modularity qualities. A response of a hierarchical FSM is specified in this semantics depth-first. The current state's deepest refinement responds first, followed by its container state machine, container, etc.

The refining machine responds first if B (which indicates that it is in either C or D) is true. The transition is made to D and action a4 is carried out if it is C and guard g4 is true. The top-level FSM then responds as a part of the same response. The machine switches to state if guard g1 is also true. It is crucial that these two transitions be logically immediate and synchronous to prevent the machine from really entering state D. However, both action a1 and action a4 are carried out.

Another nuance is that two (non-absent) acts may clash if they are carried out in the same response. Two actions, for instance, could write different values to the same output port. Alternately, they can assign several values to the same variable. Our decision is that the acts are carried out sequentially, as the semicolon in the action a4; a1 suggests. A series is indicated with a semicolon, much as in an imperative language like C.

The guards of a preemptive transition are assessed before the refinement responds, and if any guard evaluates to true, the refinement does not react which has the semantics As a result, neither operation a3 nor a4 is carried out if the machine is in state B and g1 is true. A (red) circle at the transition's originating end signifies a preemptive transition regardless of whether the machine was in D when it left B, whenever it enters B, it always enters C. The destination refinement is reset to its starting state, regardless of where it was before, during the transition from A to B, which is why it is known as a reset transition. In our notation, a reset transition is denoted by a hollow arrowhead at the final position of the transition.

A changeover in history is indicated by a solid arrowhead in our notation. For emphasis, it might alternatively be denoted with a "H". The destination refinement restarts in the state it was last in when a historical transition is made (or its initial state on the first entry) to C. It will be in the state denoted by (B, C) the first time it travels to state B, indicating that it is in state B and, more particularly, C. If it then switches from state A to state (A, D), it will remain in state A, but if and when it next reaches state (B, D), it will switch to state (B, D). In other words, it can recall its past, particularly where it was when it departed B. Similar to concurrent composition, hierarchical state machines allow for a wide range of interpretations. The variations might be slight. To make sure that models are understandable and that their semantics correspond to the data being represented, much effort must be taken.

CONCLUSION

The composition of state machines is a powerful technique for building complex systems from smaller, more manageable components. By breaking down a system into smaller state machines, developers can focus on designing the behavior of individual components and then combine them in various ways to create more complex behaviors. One key advantage of this approach is that it promotes modularity and code reuse. By designing state machines that encapsulate specific functionality, developers can reuse those machines in different contexts or even across different projects. This not only saves time but also reduces the likelihood of introducing bugs or errors that can arise when building complex systems from scratch. Another advantage of the composition of state machines is that it allows for a more flexible and dynamic system. By combining smaller state machines into larger ones, developers can create systems that can adapt to changing circumstances or user inputs. This is especially important in domains such as robotics, where systems need to be able to respond quickly to changes in their environment. The composition of state machines is a valuable technique for building complex systems that are modular, reusable, and flexible. While it requires some upfront planning and design, the benefits of this approach can be significant in terms of development time, system reliability, and maintainability.

REFERENCES:

[1] S. Beidu, J. M. Atlee, and P. Shaker, "Incremental and commutative composition of state-machine models of features," in *Proceedings - 7th International Workshop on Modeling in Software Engineering, MiSE 2015*, 2015. doi: 10.1109/MiSE.2015.10.

[2] E. V. Shirokova, S. A. Prokopenko, and N. V. Shabaldina, "On deriving the parallel composition of extended finite state machines," *Vestn. Tomsk. Gos. Univ. - Upr. Vychislitel'naya Tekhnika i Inform.*, 2019, doi: 10.17223/19988605/48/10.

[3] J. Kufner and R. Marik, "Restful State Machines and SQL Database," *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2944807.

[4] P. Wijesinghe, G. Srinivasan, P. Panda, and K. Roy, "Analysis of liquid ensembles for enhancing the performance and accuracy of liquid state machines," *Front. Neurosci.*, 2019, doi: 10.3389/fnins.2019.00504.

[5] Q. Li *et al.*, "A state machine control based on equivalent consumption minimization for fuel cell/ supercapacitor hybrid tramway," *IEEE Trans. Transp. Electrif.*, 2019, doi: 10.1109/TTE.2019.2915689.

[6] R. Lakshman Naika, R. Dinesh, and S. Prabhanjan, "Handwritten electric circuit diagram recognition: An approach based on finite state machine," *Int. J. Mach. Learn. Comput.*, 2019, doi: 10.18178/ijmlc.2019.9.3.813.

[7] A. Tvardovskii and N. Yevtushenko, "ON PARALLEL COMPOSITION OF FINITE STATE MACHINES WITH TIMED GUARDS," *Syst. Informatics*, 2019, doi: 10.31144/si.2307-6410.2019.n14.p55-64.

[8] R. Xiao, Z. Wu, and D. Wang, "A Finite-State-Machine model driven service composition architecture for internet of things rapid prototyping," *Futur. Gener. Comput. Syst.*, 2019, doi: 10.1016/j.future.2019.04.050.

[9] P. Sundar, R. Raju, P. Mahalakshmi, and G. Lalitha, "Providing security for Web Service Composition using Finite State Machine," *Int. J. Comput. Technol.*, 2005, doi: 10.24297/ijct.v4i2b1.3232.

[10] C. J. Bartel, A. Trewartha, Q. Wang, A. Dunn, A. Jain, and G. Ceder, "A critical examination of compound stability predictions from machine-learned formation energies," *npj Comput. Mater.*, 2020, doi: 10.1038/s41524-020-00362-y.

CHAPTER 6

HARDWARE VERIFICATION USING A SIMULATOR

Mrs. Aruna Dore, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email Id- aruna.dore@presidencyuniversity.in

ABSTRACT:

Hardware verification is an essential step in the design and development of complex digital circuits. In recent years, simulators have become increasingly popular for verifying hardware designs. This paper presents an overview of hardware verification using a simulator, including the benefits of using a simulator, the different types of simulators available, and some of the key features and capabilities of modern simulators. The paper discusses the role of simulators in hardware design and development, including the use of simulators for functional verification, performance analysis, and hardware-in-the-loop testing. The advantages of using a simulator, such as the ability to simulate complex circuits, reduce design costs, and improve time-to-market, are highlighted.

KEYWORDS:

Complex Circuit, Digital Circuits, Hardware Verification, Simulator.

INTRODUCTION

Hardware verification is a critical process in the design and development of electronic systems. It is essential to ensure that the hardware is working as intended, free of bugs and errors, and meets the requirements of the user or the client. Verification is a time-consuming and challenging task, especially for complex systems that have millions of transistors and are subjected to various environmental factors, such as temperature, voltage, and noise.Hardware verification using a simulator is a method that allows engineers to test and validate the design of the hardware before it is fabricated. It involves creating a virtual environment that simulates the behavior of the hardware and enables the engineers to evaluate its performance, functionality, and reliability. The simulator executes the hardware description language (HDL) code, which is the language used to describe the design, and provides a model of the circuit that can be analyzed, debugged, and improved [1].

Hardware Verification Process

The hardware verification process typically involves the following steps:

1. Specification: In this step, the requirements of the system are defined, and the specifications are written. The specification includes the functional requirements, performance goals, and design constraints.
2. Design: In this step, the hardware is designed using an HDL, such as Verilog or VHDL. The design is usually divided into modules, each with its own functionality.
3. Simulation: In this step, the design is simulated using a simulator. The simulator executes the HDL code and generates waveforms that represent the behavior of the hardware.

4. Debugging: In this step, the waveforms are analyzed, and any bugs or errors are identified. The design is then modified to fix the bugs.
5. Verification: In this step, the modified design is verified to ensure that the bugs have been fixed, and the hardware meets the specifications.

Hardware Verification Using a Simulator

Hardware verification using a simulator is a process that involves simulating the design of the hardware using a software tool. The simulator takes the HDL code as input and generates a model of the hardware, which is then simulated using test vectors. The simulation generates waveforms that represent the behavior of the hardware, and these waveforms can be analyzed to determine the correctness of the design [2]. There are two main types of simulators: event-driven and cycle-based. Event-driven simulators are more commonly used in the industry as they are more efficient and handle large designs better. Event-driven simulators operate by modeling the hardware as a collection of events. An event is a change in the state of a signal or a register. The simulator maintains a queue of events that are scheduled to occur, and it processes them in the order they occur. When an event occurs, the simulator updates the state of the hardware and schedules new events based on the new state[3], [4]. Cycle-based simulators, on the other hand, operate by modeling the hardware as a set of clock cycles. The simulator performs one clock cycle at a time, and it updates the state of the hardware at the end of each cycle. Cycle-based simulators are less efficient than event-driven simulators and are typically used for smaller designs.

The following are the steps involved in hardware verification using a simulator:

1. HDL Code Development: The first step is to write the HDL code for the design. This involves defining the modules and their functionalities, as well as their inputs and outputs. The HDL code is written in a text editor and saved as a file.
2. Simulation Setup: The next step is to set up the simulation environment. This involves creating a testbench, which is a module that generates the input signals for the design and monitors the output signals. The testbench is also responsible for running the simulation and collecting the results.
3. Simulation Execution: The simulation is then executed using the simulator tool. The simulator reads the HD

L code and the testbench file and generates a model of the hardware. The simulator then executes the testbench, which generates the input signals for the hardware and monitors the output signals. The simulator captures the behavior of the hardware in the form of waveforms, which represent the values of the signals over time.

4. Waveform Analysis: Once the simulation is complete, the waveforms are analyzed to determine the correctness of the design. The waveforms can be viewed using a waveform viewer, which is a software tool that displays the waveforms graphically. The waveform viewer allows the engineer to zoom in on specific areas of the waveform and analyze the behavior of the signals in detail.
5. Debugging: If any bugs or errors are found in the design, they can be debugged using the waveform viewer. The engineer can examine the waveforms to determine the cause of the

problem and modify the HDL code to fix it. The simulation can then be rerun to verify that the bug has been fixed.

6. Verification: Once the design has been debugged, it can be verified to ensure that it meets the specifications. Verification involves running a set of test cases that cover all the possible scenarios that the hardware is likely to encounter in operation. The test cases are executed using the testbench, and the results are analyzed using the waveform viewer.

Advantages of Hardware Verification Using a Simulator

Hardware verification using a simulator has several advantages over other methods of verification:

1. Time-saving: Hardware verification using a simulator is much faster than traditional methods of verification, such as breadboarding and prototype development. The simulation can be run in a matter of minutes or hours, while building a prototype can take days or weeks.
2. Cost-effective: Hardware verification using a simulator is also more cost-effective than building prototypes. The cost of fabrication and assembly of a prototype can be significant, while the cost of running a simulation is relatively low.
3. Early detection of bugs: Hardware verification using a simulator allows engineers to detect and fix bugs early in the design process, before the hardware is fabricated. This reduces the cost and time associated with fixing bugs after the hardware is fabricated.
4. Design optimization: Hardware verification using a simulator allows engineers to optimize the design of the hardware before it is fabricated. The simulator can be used to test different design alternatives and evaluate their performance and reliability.

Before the maritime demonstration, a simulation verification of the hybrid propulsion ship taking into account an accurate electro-dynamic model is necessary to assess ship maneuverability. In terms of the dynamic models correctness and the enhancement of a calculation speed, this research offers the streamlined model constructions for the hybrid propulsion ship employing a real-time simulator. In addition, it is suggested to use the produced equipment and a hardware-in-the-loop simulator based on a virtual instrument engineering workbench in a lab to assess the electrodynamics properties of hybrid propulsion ship. Through simulation and comparison to a comprehensive model on an electronic circuit simulation programme, the validity of a given model on a hardware in the loop simulator is confirmed[5]–[7].

A brand-new islanding detection method (IDT) based on the periodic maxima of superimposed voltage components and the islanding discrimination factor is suggested. The half-angle theorem is used to derive the sequence components of the voltage profile of the target distributed generator, which is continually monitored. The IDT technique, which is aimed for field-programmable gate arrays (FPGAs), is implemented using a modular approach. Language for Hardware Description Verilog has been used to reduce computing complexity and optimise hardware resources. The proposed IDT has been hardware-in-loop verified for islanding and non-islanding events using a microgrid (MG) test system created on a real-time digital simulator. On the created MG system, several scenarios for both islanding and non-islanding occurrences, i.e., various faults, are realised. The findings imply that the suggested approach properly identifies the islanding event under perfect power balancing conditions within half a cycle and effectively distinguishes islanding occurrences from a non-islanding event [8].

At the Institute of Space Systems at the German Aerospace Center (DLR) in Bremen, Germany, the Facility for Attitude Control Experiments (FACE) has been put into place. The facility's goal is to test and validate the Attitude Determination and Control System (ADCS) in a low Earth orbit environment. A thorough end-to-end verification of ADCS, which takes into account its growing software and hardware, is crucial to the mission's success. A hemispherical air-bearing platform with satellite component, an early solar simulator, and a magnetic field simulator make up FACE [9].

Additionally, the platform's whole centre of mass (CoM) is adjusted by the automated centre of mass calibration software, allowing it to move with the least amount of friction in all three axes as if it were in orbit. The platform also offers a WLAN command interface and a multi-output power distribution system. As a result, a self-sustaining satellite that may undergo numerous testings is set up on the platform. All that is required to do a hardware-in-the-loop test utilising flying components is to place the relevant sensor or actuator onto the platform. In particular, FACE is now ready to verify the CompactSatellite programme of the DLR. We present the real-time test bench together with certain devices, their capabilities, and upcoming enhancements. Key Words: Hardware-in-the-Loop, Small Satellites, Air-Bearing Table Nomenclature, Attitude Determination and Control.

The elliptic curve digital signature technique (ECDSA), which is useful for validating transactions in Blockchain-related applications, in hardware form. Despite the computationally costly nature of the ECDSA design, quick execution of arithmetic operations is made possible by the use of a specialised standalone circuit. For each elliptic curve, the proposed prototype provides N-bit elliptic curve cryptography (ECC) group operations, signature creation, and verification over a prime field. The study suggests a novel hardware architecture for modular multiplication and its modular multiplicative inverse, which is used for ECDSA's group operations. The register transfer logic (RTL) simulator provided by modelsim is used to simulate each hardware design. To demonstrate the superiority of the proposed work, the field programmable gate array (FPGA) implementation of several modules within the ECDSA circuit is contrasted with comparable current approaches that are based on both hardware and software.

The majority of the trams that have lately been launched use an independently rotating wheelset to achieve the tram's low-floor configuration. Trains operating in two areas with different gauges may make use of the variable track technology by using an independently rotating wheelset. The difference in rotational speed between the outer and inner wheels naturally occurs during curved driving because the independent rotation type wheelset has no rotational restraint of the left and right wheels. Because it smoothly drives curved driving, it is applied to railroad vehicles travelling in sharp curve sections. However, when the left and right rotational limitations are removed, the longitudinal creep force and the lateral restoring force are reduced. Stability is compromised by a lack of lateral direction restoring force, which results in a zigzag or continuous flange contact driving phenomena.

Verifying the Jena-Optronik laser-based sensors used by the Automated Transfer Vehicle was one of the numerous contributions the former EPOS 1.0 facility provided to spaceflight rendezvous. EPOS 2.0 is an entirely new architecture that targets far more complicated rendezvous circumstances even if it draws on its history. The On-Orbit-Servicing & Autonomy group at GSOC, which manages, develops, and executes EPOS 2.0, has achieved a lot of advancements in the area of uncooperative rendezvous during the last 10 years, with EPOS serving as its major instrument. Following general optical navigation research in the early 2010s, the DLR project "On-Orbit-Servicing End-to-End Simulation" saw the OOS group assume a leadership position in 2014. The hardware in the loop simulator of the rendezvous phase, EPOS 2.0, played a significant role in the project's resounding success. The severe needs that E2E has exposed over time have prompted multiple facility expansions and enhancements. The capabilities of EPOS 2.0 have been extensively impacted by the OOS group's research as well as many and varied open-loop test campaigns for industry and internal (DLR) clients.

DISCUSSION

Hardware verification using a simulator is a powerful method for verifying the design of electronic systems. It allows engineers to test the design before it is fabricated, detect bugs and errors, and optimize the design for performance and reliability. The simulation process involves developing HDL code, setting up the simulation environment, executing the simulation, analyzing the waveforms, debugging the design, and verifying that the hardware meets the specifications. Hardware verification using a simulator is faster, more cost-effective, and enables early detection of bugs, making it an essential part of the hardware development process.

Hardware verification using a simulator has become an essential part of the hardware design process. As the complexity of electronic systems increases, it becomes increasingly difficult and expensive to verify the design using traditional methods. Simulation offers a faster, more cost-effective, and more accurate way to test the design of the hardware.

Simulation tools have evolved significantly in recent years, and modern simulators offer a range of features and capabilities that make them powerful tools for hardware verification. For example, some simulators can perform advanced analysis of the waveforms, including statistical analysis, waveform comparison, and waveform alignment. This can help engineers to identify subtle problems in the design that might be difficult to detect using a waveform viewer alone. Simulation tools can also be used to generate test cases automatically. By analyzing the HDL code and the test bench, the simulator can create a set of test cases that cover all the possible scenarios that the hardware is likely to encounter. This can save engineers a significant amount of time and effort in developing test cases manually[10].

Another advantage of hardware verification using a simulator is that it can be used to verify the design of complex systems that would be difficult or impossible to verify using traditional methods. For example, simulators can be used to verify the design of systems that contain multiple processors, complex communication protocols, or large amounts of memory. One of the challenges of hardware verification using a simulator is the need for a high level of expertise in the use of the simulation tools. Setting up the simulation environment, developing the HDL code, and analyzing the waveforms requires a significant amount of skill and experience. In addition, simulators can be expensive, and the cost of the simulation tools can be a barrier to entry for smaller companies or individual designers.

In conclusion, hardware verification using a simulator is a powerful and essential tool for verifying the design of electronic systems. It offers a range of advantages over traditional methods, including speed, cost-effectiveness, and early detection of bugs. While there are challenges associated with using simulators, the benefits of this approach make it a valuable part of the hardware development process.

Another advantage of hardware verification using a simulator is that it enables engineers to perform what-if analysis on the design. By changing the parameters of the HDL code, such as clock frequencies or data widths, engineers can evaluate the impact of these changes on the performance and reliability of the hardware. This can help engineers to optimize the design and identify the best set of parameters for the system.

Hardware verification using a simulator can also be used to verify the timing of the design. Timing verification is critical for many applications, especially those that involve high-speed data transfer or processing. Simulators can be used to verify the timing of the design at various clock frequencies and operating conditions, ensuring that the hardware meets the required timing specifications. Simulators can also be used to verify the power consumption of the hardware. Power consumption is a critical parameter in many applications, especially those that are battery-powered or that require low power consumption. Simulators can be used to estimate the power consumption of the hardware under different operating conditions, enabling engineers to optimize the design for power consumption.

Finally, hardware verification using a simulator can be used to develop virtual prototypes of the hardware. A virtual prototype is a software representation of the hardware that can be used to test and verify the design. Virtual prototypes can be used to evaluate the performance and functionality of the hardware without the need for physical prototypes. This can save time and cost in the development process, especially for complex systems.

In conclusion, hardware verification using a simulator is a powerful and essential tool for verifying the design of electronic systems. It offers a range of advantages over traditional methods, including speed, cost-effectiveness, and early detection of bugs. It also enables engineers to perform what-if analysis, verify the timing and power consumption of the design, and develop virtual prototypes of the hardware. While there are challenges associated with using simulators, the benefits of this approach make it a valuable part of the hardware development process. In Figure 1 illustrate the simulation-based hardware verification.

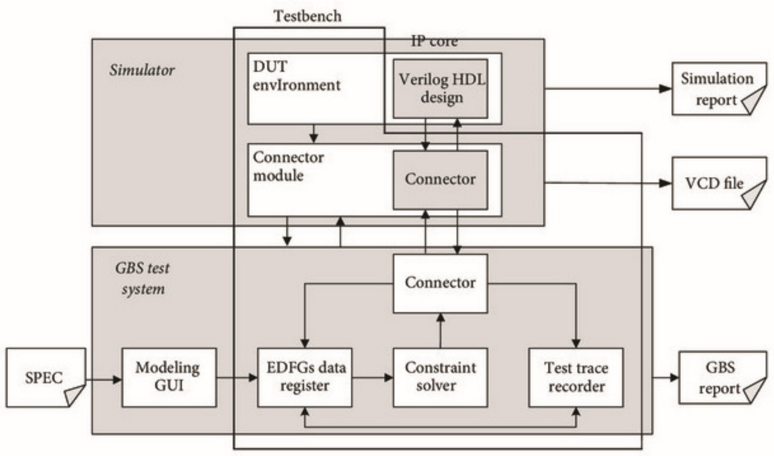


Figure 1: Illustrate the Simulation-Based Hardware Verification.

Hardware verification is an essential part of the hardware design process. It involves testing the hardware design to ensure that it meets the required specifications and performs as expected. Traditionally, hardware verification was done using physical prototypes, but this approach is time-consuming, expensive, and often impractical for complex systems. Hardware verification using a simulator offers a faster, more cost-effective, and more accurate way to test the design of the hardware.

A simulator is a software tool that creates a model of the hardware design in software. The simulator can then be used to test the design using a testbench, which is a collection of test cases designed to test the functionality of the hardware. The testbench sends input signals to the hardware design, and the simulator generates output signals based on the behavior of the design. The output signals are then compared to the expected results to determine whether the hardware design is functioning as expected.

One of the primary advantages of hardware verification using a simulator is that it is faster and more cost-effective than traditional methods. Testing a hardware design using physical prototypes can be time-consuming and expensive, especially for complex systems that require multiple prototypes. In contrast, a simulator can be run on a computer, enabling engineers to test the design much more quickly and at a lower cost. This can significantly reduce the time and cost of the hardware development process, enabling engineers to get their products to market more quickly and at a lower cost.

Hardware verification using a simulator is also more accurate than traditional methods. Simulators can provide detailed information about the behavior of the hardware design, including waveforms, timing diagrams, and statistical information. This can help engineers to identify subtle problems in the design that might be difficult to detect using traditional methods. In addition, simulators can generate large amounts of data that can be analyzed to identify patterns and trends in the behavior of the hardware design. In Figure 2 illustrate the validation and verification for system development. In Figure 2 shows the validation for system development.

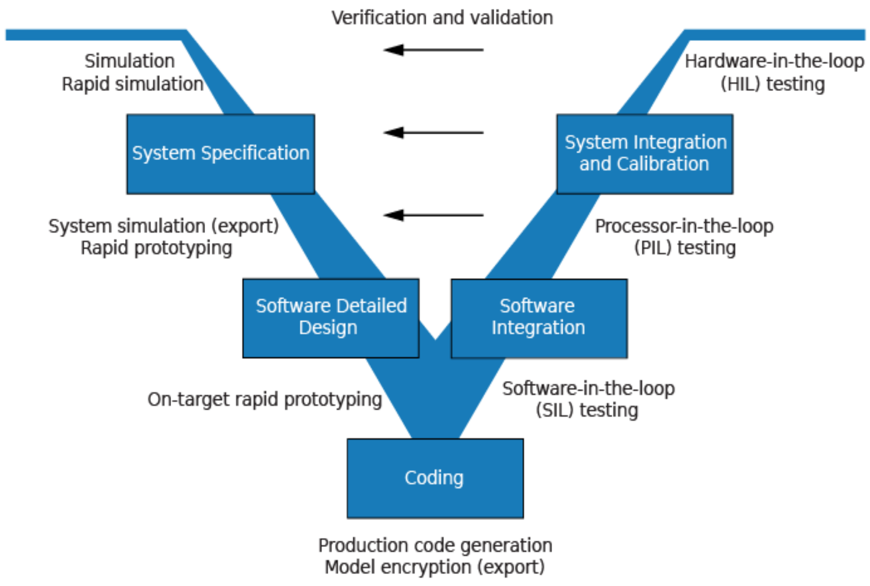


Figure 2: Illustrate the validation and verification for system development.

Simulators can also be used to perform what-if analysis on the hardware design. What-if analysis involves changing the parameters of the design, such as clock frequencies or data widths, to evaluate the impact of these changes on the performance and reliability of the hardware. This can help engineers to optimize the design and identify the best set of parameters for the system. What-if analysis is particularly useful for complex systems that have multiple parameters that can be adjusted to optimize performance.

Hardware verification using a simulator can also be used to verify the timing of the design. Timing verification is critical for many applications, especially those that involve high-speed data transfer or processing. Simulators can be used to verify the timing of the design at various clock frequencies and operating conditions, ensuring that the hardware meets the required timing specifications. Simulators can also be used to verify the power consumption of the hardware. Power consumption is a critical parameter in many applications, especially those that are battery-powered or that require low power consumption. Simulators can be used to estimate the power consumption of the hardware under different operating conditions, enabling engineers to optimize the design for power consumption.

Finally, hardware verification using a simulator can be used to develop virtual prototypes of the hardware. A virtual prototype is a software representation of the hardware that can be used to test and verify the design. Virtual prototypes can be used to evaluate the performance and functionality of the hardware without the need for physical prototypes. This can save time and cost in the development process, especially for complex systems.

One of the challenges of hardware verification using a simulator is the need for a high level of expertise in the use of the simulation tools. Setting up the simulation environment, developing the HDL code, and analyzing the waveforms requires a significant amount of skill and experience. In addition, simulators can be expensive, and the cost of the simulation tools can be a barrier to entry for smaller companies or individual designers design of complex hardware systems. The benefits of using a simulator include faster and more cost-effective testing, more accurate results, what-if analysis, timing and power consumption verification, and the ability to develop virtual prototypes. However, it is important to recognize that using a simulator requires a high level of expertise, and the cost of the simulation tools can be a barrier to entry.

To begin the process of hardware verification using a simulator, the first step is to create a model of the hardware design in software. This is typically done using a Hardware Description Language (HDL) such as Verilog or VHDL. The HDL code describes the behavior of the hardware design, including the inputs and outputs, logic gates, and other components. The HDL code is compiled using a software tool that generates a model of the design that can be used with a simulator.

The next step is to create a testbench for the hardware design. The testbench is a collection of test cases that are designed to test the functionality of the hardware design. The testbench sends input signals to the hardware design, and the simulator generates output signals based on the behavior of the design. The output signals are then compared to the expected results to determine whether the hardware design is functioning as expected. Testbench development is a critical part of the verification process, and it is important to develop a comprehensive set of test cases that cover all of the functionality of the hardware design. The testbench should be designed to test the design under different operating conditions, including different clock frequencies, input signal patterns, and environmental conditions.

Once the testbench is developed, the next step is to run the simulation. The simulator generates waveforms that show the behavior of the hardware design over time. The waveforms can be analyzed to determine whether the design is functioning as expected. In addition, the simulator can generate timing diagrams that show the timing relationships between the various signals in the design. The timing diagrams can be used to verify that the design meets the required timing specifications.

The output of the simulation can be analyzed in several ways. First, the waveforms can be analyzed to identify any unexpected behavior in the design. This can include glitches, timing violations, or other issues that could affect the performance of the design. Second, the timing diagrams can be analyzed to verify that the design meets the required timing specifications. Finally, statistical analysis can be performed on the output of the simulation to identify any patterns or trends in the behavior of the design.

What-if analysis is another powerful tool that can be used in hardware verification using a simulator. What-if analysis involves changing the parameters of the design, such as clock frequencies or data widths, to evaluate the impact of these changes on the performance and reliability of the hardware. This can help engineers to optimize the design and identify the best set of parameters for the system.

Timing verification is critical for many applications, especially those that involve high-speed data transfer or processing. Simulators can be used to verify the timing of the design at various clock frequencies and operating conditions, ensuring that the hardware meets the required timing specifications. This is particularly important for applications such as high-speed data communication, where timing errors can result in lost data or other issues.

Power consumption is another critical parameter in many applications, especially those that are battery-powered or that require low power consumption. Simulators can be used to estimate the power consumption of the hardware under different operating conditions, enabling engineers to optimize the design for power consumption.

In addition to functional verification, simulators can also be used for verification of the physical design of the hardware. This includes checking for physical design issues such as routing congestion, timing issues, and other problems that could affect the performance of the design. Physical design verification is typically done using a separate tool such as a place-and-route tool.

CONCLUSION

Hardware verification using a simulator is a critical part of the hardware design process. Simulators allow engineers to test the functionality, timing, and power consumption of the design in a virtual environment, which can be faster and more cost-effective than testing in hardware. Simulators also allow for what-if analysis, enabling engineers to optimize the design for performance and reliability. The use of a simulator requires a high level of expertise and the cost of the simulation tools can be a barrier to entry. However, the benefits of using a simulator, including faster and more cost-effective testing, more accurate results, and the ability to develop virtual prototypes, make it a valuable tool for hardware design.

REFERENCES:

[1] C. Klug, D. Schmalstieg, T. Gloor, and C. Arth, “A Complete Workflow for Automatic Forward Kinematics Model Extraction of Robotic Total Stations Using the Denavit-Hartenberg Convention,” *J. Intell. Robot. Syst. Theory Appl.*, 2019, doi: 10.1007/s10846-018-0931-4.

[2] S. Raikwar, L. Jijyabhau Wani, S. Arun Kumar, and M. Sreenivasulu Rao, “Hardware-in-the-Loop test automation of embedded systems for agricultural tractors,” *Meas. J. Int. Meas. Confed.*, 2019, doi: 10.1016/j.measurement.2018.10.014.

[3] S. Yang and Z. Yu, “A Highly Integrated Hardware/Software Co-Design and Co-Verification Platform,” *IEEE Des. Test*, 2019, doi: 10.1109/MDAT.2018.2841029.

[4] M. Qin, W. Hu, X. Wang, D. Mu, and B. Mao, “Theorem proof based gate level information flow tracking for hardware security verification,” *Comput. Secur.*, 2019, doi: 10.1016/j.cose.2019.05.005.

[5] A. Lööw *et al.*, “Verified compilation on a verified processor,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, 2019. doi: 10.1145/3314221.3314622.

[6] B. Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, “Instruction-level abstraction (ILA): A uniform specification for system-on-chip (SOC) verification,” *ACM Trans. Des. Autom. Electron. Syst.*, 2019, doi: 10.1145/3282444.

[7] E. Gezgin, S. Özbek, D. Güzin, O. E. Ağbaş, and E. B. Gezer, “Structural design of a positioning spherical parallel manipulator to be utilized in brain biopsy,” *Int. J. Med. Robot. Comput. Assist. Surg.*, 2019, doi: 10.1002/rcs.2011.

[8] P. Kumar, V. Kumar, and R. Pratap, “FPGA implementation of an Islanding detection technique for microgrid using periodic maxima of superimposed voltage components,” *IET Gener. Transm. Distrib.*, 2020, doi: 10.1049/iet-gtd.2018.5914.

[9] T. KATO, A. HEIDECKER, M. DUMKE, and S. THEIL, “Three-Axis Disturbance-Free Attitude Control Experiment Platform : FACE,” *Trans. JAPAN Soc. Aeronaut. Sp. Sci. Aerosp. Technol. JAPAN*, 2014, doi: 10.2322/tastj.12.td_1.

[10] H. Gao, H. Yu, G. Xie, H. Ma, Y. Xu, and D. Li, “Hardware and software architecture of intelligent vehicles and road verification in typical traffic scenarios,” *IET Intell. Transp. Syst.*, 2019, doi: 10.1049/iet-its.2018.5351.

CHAPTER 7
COMPOSITION OF STATE MACHINES

Mrs. Manaswini R, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email id- manaswini.r@presidencyuniversity.in

ABSTRACT:

Composition of state machines is a fundamental concept in the field of computer science, particularly in the design of complex systems. State machines are used to model the behavior of a system, where the system is represented as a collection of states and transitions. In the composition of state machines, multiple state machines are combined to form a more complex system, with the behavior of the system determined by the behavior of the individual state machines and the interactions between them. This approach enables the creation of complex systems that can be broken down into simpler components, making it easier to analyze and verify their behavior. This paper provides an overview of the composition of state machines, including the principles of state machine composition, the benefits and challenges of using this approach, and practical applications in the design of computer systems.

KEYWORDS:

Complex System, Components, Design, State Machine, Transitions.

INTRODUCTION

State machines, also known as finite-state machines, are computational models that represent the behavior of a system as a set of states and transitions between them. They are widely used in various domains, such as software engineering, control systems, and digital circuits. State machines can be composed to form more complex systems, which is a common technique in the design of large and complex systems. In this article, we will discuss the composition of state machines, including its benefits, challenges, and techniques [1].

The composition of state machines allows the creation of more complex systems from simpler ones, which provides several benefits, such as: The modularity and reusability are two of the main advantages of state machine composition. By breaking down a complex system into simpler state machines, it is possible to reuse the simpler components in other systems, leading to faster development, easier maintenance, and better scalability. This approach can also improve the design of the system by enforcing a clear separation of concerns, leading to better overall organization.

Another benefit of state machine composition is that it enables incremental development. By developing individual state machines independently, it is possible to integrate them gradually into a larger system, which reduces the risk of errors and makes the development process more manageable. State machine composition also provides flexibility and adaptability. By creating state machines that can be easily combined and extended, it is possible to adapt the system to changing requirements or new features without major changes to the existing codebase. This approach can also make the system more robust and fault-tolerant by allowing different parts of the system to fail without affecting the entire system [2].

Although state machine composition provides several benefits, it also presents several challenges that need to be addressed, such as:

One of the main challenges of state machine composition is state explosion. As the number of states and transitions in a system increases, the complexity of the system also increases, which can lead to combinatorial explosion. This can make the system difficult to understand and test, leading to errors and bugs that are hard to find. Another challenge of state machine composition is coordination and synchronization. As state machines are combined, it is necessary to ensure that they work together correctly and that their states and transitions are synchronized. This requires careful design and testing to avoid conflicts and inconsistencies.

The design of the interface between state machines is another challenge of state machine composition. The interface needs to be well-defined and consistent to ensure that the state machines can communicate and work together correctly. This requires a good understanding of the behavior of each state machine and the requirements of the system as a whole. Several techniques can be used for state machine composition, each with its strengths and weaknesses. Hierarchical composition is a technique that involves creating a hierarchy of state machines, where each state machine can contain other state machines as subcomponents. This approach can reduce the complexity of the system by breaking it down into smaller, more manageable components. It also enables reuse of the subcomponents in other systems. However, hierarchical composition can lead to problems with synchronization and coordination, especially if the subcomponents need to communicate with each other.

Parallel composition is a technique that involves combining multiple state machines in parallel, where each state machine runs independently and communicates with the others as needed. This approach can provide better performance and scalability than hierarchical composition, as it enables the system to take advantage of multiple processors or cores. However, parallel composition can also lead to problems with synchronization and consistency, especially if the state machines need to access shared resources [3]. State machine refinement is a technique that involves decomposing a state machine into smaller, more manageable state machines. This approach can be useful for reducing the complexity of the system and improving its modularity and reusability. State machine refinement can be done using several methods, such as state decomposition, state aggregation, and state abstraction.

State decomposition involves breaking down a state machine into smaller state machines that represent a subset of the original states and transitions. This approach can make the system more modular and easier to understand, as each subcomponent can be developed and tested independently. State decomposition can also enable the reuse of the subcomponents in other systems. State aggregation involves combining several state machines into a larger state machine that represents the behavior of the system as a whole. This approach can be useful for simplifying the design of the system and reducing the number of states and transitions. State aggregation can also enable the reuse of the state machines in other systems.

State abstraction involves replacing a complex state or set of states with a simpler state or set of states that represents the same behavior. This approach can be useful for reducing the complexity of the system and making it easier to understand and test. State abstraction can also enable the reuse of the simplified state machines in other systems.

The design of the interface between state machines is an important aspect of state machine composition. The interface needs to be well-defined and consistent to ensure that the state machines can communicate and work together correctly. Several approaches can be used for interface design, such as synchronous communication, asynchronous communication, and shared memory.

Synchronous communication involves the exchange of messages between state machines in a coordinated manner, where each state machine waits for a response from the other before proceeding. This approach can be useful for ensuring consistency and synchronization between state machines, but it can also lead to blocking and deadlock if the communication is not well-designed. Asynchronous communication involves the exchange of messages between state machines without waiting for a response from the other. This approach can be useful for reducing blocking and enabling more parallelism in the system, but it can also lead to synchronization and consistency issues if the communication is not well-designed[4].

Shared memory involves the use of a shared data structure that can be accessed by multiple state machines. This approach can be useful for enabling communication and synchronization between state machines, but it can also lead to conflicts and race conditions if the access to the shared memory is not properly managed.

State machine composition is a powerful technique for building complex systems from simpler state machines. It provides several benefits, such as modularity, reusability, incremental development, flexibility, and adaptability. However, state machine composition also presents several challenges, such as state explosion, coordination and synchronization, and interface design. Several techniques can be used for state machine composition, such as hierarchical composition, parallel composition, and state machine refinement, each with its strengths and weaknesses. The design of the interface between state machines is also an important aspect of state machine composition and requires careful consideration to ensure that the state machines can communicate and work together correctly.

An unprecedented possibility for a broad range of applications is provided by the developing area of the Internet of Things (IoT). The majority of apps, however, have been using closed technology stacks and proprietary ways to integrate IoT devices. High customisation costs and constrained component reusability are caused by the monolithic, mostly vendor-specific development architecture. Full-fledged IoT applications are hampered in cross-organizational, all-purpose, and dynamically changing circumstances. With the help of this study, IoT fast prototyping will be made interoperable, affordable, and user-customizable. Each IoT component, whether it be a physical device or control logic, is abstracted into a separate web service under this architecture, which is characterised by a collection of transferable states. IoT components are further put together into adaptable apps by concatenating a legitimate chain of state transfers across web services [5].

A high-entropy alloy is a major component-free alloy design idea. This idea implies that there will be more intermediate metastable states when the high-entropy alloy transitions from a high-energy state to a low-energy one, in addition to referring to the complexity of alloy compositions. Changes in the microstructure's degree and kind of order correspond to distinct states. In this work, we combined elemental features with long-term ordering using machine learning, and we produced 87% prediction accuracy. This data-driven approach may speed up the identification of prospective compositions by correlating elemental properties with metastable states.

For state machines and the composition of them in UML, we describe a compositional operational semantics. Each state machine explains how a class of item behaves. A new activity group, which is a single-threaded collection of objects, is formed if a class of a newly generated object is active. State machine communication between activity groups is different from that inside an activity group. We offer I two parallel combinatory that reflect this distinction and that, if their arguments are SOS, return SOSs as well as (ii) an SOS for each state machine taken separately [6].

Machine learning has quickly changed the way that materials science predicts their properties. The limited development of techniques to learn the underlying interactions of multiple elements as well as the relationships among multiple properties to facilitate property prediction in new composition spaces is one obstacle preventing full capitalization of recent advancements in machine learning. In order to solve these problems, we present the Hierarchical Correlation Learning for Multi-property Prediction (H-CLMP) framework, which seamlessly combines the following three methods: (i) prediction using just the composition of a material; (ii) learning and exploitation of correlations among target properties in multi-target regression; and (iii) leveraging training data from tangential domains via generative transfer learning. It is shown how the model predicts the spectrum optical absorption of 69 different three-cation metal oxide composition spaces for complicated metal oxides.

The relevance of security offered by Web Services during Web Service Composition is highlighted by the revolution caused by Web Services as a solution to business and corporate application integration. The dynamic and erratic character of the Web makes it a difficult undertaking to satisfy the security standards. In order to develop high level business processes that perfectly fit and adhere to the demands of the service requestor, web services are combined in a process called web service composition. It entails identifying, absorbing, and delivering basic services in order to customise services often [7].

It has been said that machine-learned models for the formation energy of compounds may approach the accuracy of Density Functional Theory, which is a revolutionary tool for the effective prediction of material characteristics (DFT). Five newly published compositional models, a baseline model based only on stoichiometry, and a structural model are among the models investigated in this study. Using the Materials Project database of DFT calculations for 85,014 different chemical compositions, we tested seven machine learning models for formation energy on stability predictions. Our results demonstrate that while formation energies can be accurately predicted, all compositional models perform poorly at predicting the stability of compounds, making them much less useful than DFT for the discovery and design of new solids. Transition between rules is prompted by the successful execution of policies that are specified in the node access points [8].

William Fitzhugh et al. In the realm of energy storage, solid-state batteries (SSBs) stand out as one of the most promising developments. However, the complicated electro-chemical processes that necessarily take place at the interface of solid-state electrolyte (SSE) particles are now limiting the development of SSBs. Moreover, there is no simple approach for dealing with these interface instabilities because of the material complexity of such systems. In this work, we analyse and create solid-state solid-electrolyte-interphases (SEI) with controllable electrochemical stabilities utilising a combined high-throughput ab initio computing and machine learning technique. Machine learning indicates that the mechanical constriction effect's capacity to stabilise a solid-state SEI is a nonconvex and nonlinear, but nonetheless predictable, function of composition. Machine learning algorithm based on random forest regression to remove irrelevant parameters and assess the association between each parameter and response parameter. Using Design of Experiment on CMOST from Computer Modeling Group, 1000 experimental designs of LSWI parameters, Reservoir & Injection Temperature, Volume Injection, Formation Water Composition, and Injection Water Composition were built using Recovery Factor as the response parameter.

DISCUSSION

State machines provide a practical method for simulating system behaviour. The fact that most fascinating systems have a very large and sometimes infinite number of states is one drawback for them. Huge state spaces can be handled by automated methods, but humans struggle more with any direct depiction of a large state space.Engineering has long held the belief that complex systems should be seen as combinations of smaller systems. State machines are used in this chapter to accomplish a variety of tasks. However, the reader should be aware that there are a variety of subtle variations in how state machines might be composed. Things that on the surface seem to be same may imply various things to different individuals. A model's notational guidelines are referred to as its syntax, while the semantics of the notation are referred to as its meaning.

The preceding example's model demonstrates the complexities of concurrent systems. First, accesses to the shared variable are atomic operations because of the interleaving semantics, which is rather difficult to enforce in reality. Second, the selection of semantics 1 in this situation makes sense since the input is sent to both component machines, therefore no input event will be lost regardless of how either component machine responds. If the two computers had separate inputs, however, this semantics would not function since requests may be overlooked. That can be avoided with the aid of semantics 2, but how should the environment go about deciding? Introduction to Embedded Systems, Lee & Seshia.

Which machine responds in the composition of state machines? What if C's response to both of the independent inputs' requests is present? What does it imply when both computers update the shared variable at the same time if we pick semantics 4 in the sidebar on page 118 to enable both machines to respond simultaneously? With an interleaving semantics, the updates are no longer atomic. Also take note that semantic 1's asynchronous composition option permits actions that do not effectively use idle computers. Consider in particular a scenario in which machine A is fulfilling customer requests while machine B is idle. Machine A will just increase pending if the nondeterministic decision causes it to respond. The idle machine is not used until B responds as a consequence of the nondeterministic decision. In actuality, semantics 1 permits actions that never make use of a machine.

In synchronous compositions, shared variables may also be employed, although complex nuances once more surface. What should happen, in instance, if two machines write to the same shared variable at the same time that one machine reads it to evaluate a guard? Do we insist on writing before reading? What happens if the transition reading the shared variable in its guard expression also writes to the same variable? One option is to use a nondeterministic ally selected synchronous interleaving semantics in which the component machines respond in any order. The drawback of this approach is that a combination of two deterministic machines may not be deterministic. The component machines respond in a defined sequence set by the environment or by another mechanism, such as priority, in a different implementation of the synchronous interleaving semantics.

Shared variable challenges, especially with asynchronous composition, are a reflection of the intrinsic difficulty of concurrency models with shared variables. Clean solutions need a more complex semantics. We shall describe the synchronous-reactive model of computing in that chapter. This model provides a synchronous composition semantics that is largely compositional. Think about the two state machines A and B that are shown in Figure 5.7. Machine A's output feeds machine B's input. Cascade composition or serial composition are two names for this kind of music.

The output port o1 from A in the diagram feeds events to the input port i2 of B. Assume that o1 has the data type V1 and that i2 has the data type V2 such that o1 may either accept values from V1 or not. V1 must then equal V2 in order for this composition to be considered legitimate. According to this, every output generated by A on port o1 is a valid input for B on port i2. The composition type is validated. We need to provide some mechanism for buffering the data transmitted from A to B if we want the construction of a cascade to be asynchronous. We postpone treatment of such asynchronous composition, when it will be provided by dataflow and process network models of computing. We will solely discuss synchronous composition for cascade systems in this chapter.

A reaction of C in the cascade structure synchronous composition is made up of reactions of both A and B, where A reacts first, producing any output that may be present, and then B reacts. The two responses are, in a sense, simultaneous and instantaneous since, logically, we see this as happening in zero time. However, they are causally connected in that the actions of A may influence those of B. The image clearly illustrates how the two machines' responses are immediate and simultaneous. When the input an is missing, the composition machine transitions from the starting state (s1, s3) to (s2, s4). C Determine which transitions are taken under what circumstances before constructing the composition machine by first creating the state space as the cross product of the state spaces of the component machines. Even though one logically results in the other, it is crucial to keep in mind that the transitions happen simultaneously.

Assume that we want to construct this using a pedestrian crossing light model similar. The input sigR of the pedestrian light may be derived from the output sigR of the traffic signal. The meaning of the composite under synchronous cascade composition. Keep in mind that risky situations, such (green, green), which is the situation where both traffic and pedestrians have a green light, are not attainable states and are thus not shown. A pedestrian crossing light model that is to be built in a synchronous cascade composition with the traffic light model is what cascade composition, in its most basic form, entails. Introduction to Embedded Systems, The semantics of a synchronous cascade composition of the pedestrian light model and the traffic light model. We do not encounter the same issues with shared variables as we did with side-by-side composition since this ordering is clearly specified. We will see that the ordering is not straightforward in more generic compositions. Machine compositions that are more complicated may be built using the fundamental building blocks of side-by-side and cascade composition.

A1 and A3 together constitute the machine B. They are a side-by-side composition. B feeds events to A2 as part of a cascade composition with B. A2 feeds events to B, while B and A2 are also a cascade composition in the reverse direction. Cycles such as Lee and Seshia's Introduction to Embedded Systems.Feedbacks like these pose the dilemma of whether machine B or machine A2 should respond first. In the next chapter, when we discuss the synchronous-reactive paradigm of computing, this paradox will be clarified. Hierarchical FSMs, which predate Statecharts (Harel, 1987), are discussed in this section. Statecharts come in a variety of forms, and there are often little semantic changes between them (von der Beeck, 1994). Here, we will choose one specific semantic version and concentrate just on some of the more straightforward elements. State refinement is the central concept of hierarchical state machines. State B has a refinement that is another FSM with two states, C and D. The machine being in state B indicates that it is either in state C or state D.

The hierarchy's significance may be recognised. The device enters state A during startup. When the guard g2 evaluates to true, the machine moves to state B, which signifies a move to state C, the refinement's starting state. When the machine makes this transition to C, it does action a2, which (depending on whether this is an extended state machine) may result in an output event or set a variable.

Then, there are two routes out of C. Guard g1 must evaluate to true for the machine to leave B and return to A. If guard g4 must also evaluate to true for the machine to leave B and go to D. What happens if both guards g1 and g4 evaluate to true is a tricky issue. Different Statechart variations may choose differently at this moment. The machine should logically end up in state A, but which of the actions a4 or a1 or both should be executed? Such nuanced inquiries contribute to explaining why there are so many diverse Statecharts variations. We choose a certain semantics because it has appealing modularity qualities. A response of a hierarchical FSM is specified in this semantics depth-first. The current state's deepest refinement responds first, followed by its container state machine, container, etc.

The refining machine responds first if B which indicates that it is in either C or D is true. The transition is made to D and action a4 is carried out if it is C and guard g4 is true. The top-level FSM then responds as a part of the same response. The machine switches to state A if guard g1 is also true. It is crucial that these two transitions be logically immediate and synchronous to prevent the machine from really entering state D. However, both action a1 and action a4 are carried out. This combination lines up with the transition at the top. Another nuance is that two (non-absent) acts may clash if they are carried out in the same response. Two actions, for instance, could write different values to the same output port. Alternately, they can assign several values to the same variable. Our decision is that the acts are carried out sequentially, as the semicolon in the action a4; a1 suggests. A series is indicated with a semicolon, much as in an imperative language like C. The later action takes precedence if the two are in disagreement.

The use of a preemptive transition, which has the semantics seen in Figure 5.16, may prevent such complications. Before a refinement responds to a preemptive transition, the guards are examined; if any guard evaluates to true, the refinement does not react. As a result, neither operation a3 nor a4 is carried out if the machine is in state B and g1 is true. A (red) circle at the transition's originating end signifies a preemptive transition. Regardless of whether the machine was in D when it left B, whenever it enters B, it always enters C. The destination refinement is reset to its starting state, regardless of where it was before, during the transition from A to B, which is why it is known as a reset transition. In our notation, a reset transition is denoted by a hollow arrowhead at the final position of the transition.

Alternative to a reset transition for the move from A to B. A changeover in history is indicated by a solid arrowhead in our notation. For emphasis, it might alternatively be denoted with an "H". The destination refinement restarts in the state it was last in when a historical transition is made or its initial state on the first entry. The machine is initially in state A, as shown by the labels (A, C), and if and when it next reaches state B, it will proceed to state B.

The semantics of the history-transitioning hierarchical state machine from Figure 5.17 to C. It will be in the state denoted by (B, C) the first time it travels to state B, indicating that it is in state B and, more particularly, C. If it then switches from state A to state (A, D), it will remain in state A, but if and when it next reaches state (B, D), it will switch to state (B, D). In other words, it can recall its past, particularly where it was when it departed B.

Similar to concurrent composition, hierarchical state machines allow for a wide range of interpretations. The variations might be slight. To make sure that models are understandable and that their semantics correspond to the data being represented, much effort must be taken. Any well-engineered system is made up of smaller components. Concurrent composition and hierarchical composition are the two types of state machine composition that we have discussed in this chapter. We offered both synchronous and asynchronous composition for concurrent composition, but we left the tale unfinished. Feedback management has been delayed until the 130 Lee & Seshia, Introduction to Embedded Systems class.

CONCLUSION

Communication through ports is necessary for asynchronous composition, but these extra techniques are not (yet) included in our model of state machines. There are important complexities even without communicating across ports since there are many semantics for asynchronous composition that are available, and each has advantages and disadvantages. A certain semantics option could be appropriate for one application but not another. The theme of the next chapter, which gives concurrent composition greater structure and answers the majority of these problems, is motivated by these complexities.

REFERENCES:

[1] S. Beidu, J. M. Atlee, and P. Shaker, "Incremental and commutative composition of state-machine models of features," in *Proceedings - 7th International Workshop on Modeling in Software Engineering, MiSE 2015*, 2015. doi: 10.1109/MiSE.2015.10.

[2] E. V. Shirokova, S. A. Prokopenko, and N. V. Shabaldina, "On deriving the parallel composition of extended finite state machines," *Vestn. Tomsk. Gos. Univ. - Upr. Vychislitel'naya Tekhnika i Inform.*, 2019, doi: 10.17223/19988605/48/10.

[3] A. Tvardovskii and N. Yevtushenko, "ON PARALLEL COMPOSITION OF FINITE STATE MACHINES WITH TIMED GUARDS," *Syst. Informatics*, 2019, doi: 10.31144/si.2307-6410.2019.n14.p55-64.

[4] H. Gao, H. Yu, G. Xie, H. Ma, Y. Xu, and D. Li, "Hardware and software architecture of intelligent vehicles and road verification in typical traffic scenarios," *IET Intell. Transp. Syst.*, 2019, doi: 10.1049/iet-its.2018.5351.

[5] R. Xiao, Z. Wu, and D. Wang, “A Finite-State-Machine model driven service composition architecture for internet of things rapid prototyping,” *Futur. Gener. Comput. Syst.*, 2019, doi: 10.1016/j.future.2019.04.050.

[6] H. Fecher, M. Kyas, W. P. de Roever, and F. S. de Boer, “Compositional Operational Semantics of a UML-Kernel-Model Language,” *Electron. Notes Theor. Comput. Sci.*, 2006, doi: 10.1016/j.entcs.2005.08.008.

[7] P. Sundar, R. Raju, P. Mahalakshmi, and G. Lalitha, “Providing security for Web Service Composition using Finite State Machine,” *Int. J. Comput. Technol.*, 2005, doi: 10.24297/ijct.v4i2b1.3232.

[8] C. J. Bartel, A. Trewartha, Q. Wang, A. Dunn, A. Jain, and G. Ceder, “A critical examination of compound stability predictions from machine-learned formation energies,” *npj Comput. Mater.*, 2020, doi: 10.1038/s41524-020-00362-y.

CHAPTER 8

CONCURRENT MODELS OF COMPUTATION

Ms. Pallabi Kakati, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email id- pallabi.kakati@presidencyuniversity.in

ABSTRACT:

Concurrent models of computation are a class of mathematical and computational models used to describe and analyze concurrent systems. These models are essential for understanding the behavior of complex systems that involve multiple interacting components or agents, such as distributed systems, parallel computing, and multi-agent systems. Concurrent models of computation use a range of formalisms, including process calculi, Petri nets, and timed automata, to describe the behavior of systems that operate concurrently. These models allow us to reason about the interaction of different components, the synchronization and communication mechanisms used, and the effect of concurrent execution on the overall system behavior.

KEYWORDS:

Concurrent Model, Computational Model, Interaction Components, Parallel Computing, Multi-agent System.

INTRODUCTION

The study of computation, which seeks to understand the limits and possibilities of computing machines, has led to the development of several models of computation. These models differ in their assumptions about the underlying technology, the way they represent computations, and the types of problems they can solve efficiently. In this essay, we will explore the most important models of computation, including the Turing machine, the lambda calculus, the von Neumann architecture, and the parallel computing model [1].

Turing Machines

The Turing machine, invented by British mathematician Alan Turing in 1936, is perhaps the most well-known model of computation. The Turing machine is a theoretical device that consists of a tape divided into cells, a head that can read and write symbols on the tape, and a set of rules that specify how the machine should move the head and modify the tape based on the current symbol.The tape is infinite in both directions, so the machine can access an unbounded amount of memory.The Turing machine is a simple yet powerful model of computation that captures the essence of what it means to compute. The Church-Turing thesis, which states that any function that can be computed by an algorithm can be computed by a Turing machine, is widely accepted as a fundamental principle of computer science.

The lambda calculus

The lambda calculus, developed by American mathematician Alonzo Church in the 1930s, is a formal system for representing and manipulating functions. The lambda calculus is based on the idea of abstraction, where a function is defined in terms of a variable that represents an argument

[2]. The lambda calculus provides a way to represent functions as mathematical objects that can be manipulated and combined like numbers.

The lambda calculus is a purely functional model of computation, meaning that it does not have the notion of state or side effects that are present in imperative programming languages. This makes the lambda calculus well-suited for reasoning about programs and proving their correctness. The lambda calculus is also the foundation of functional programming languages such as Haskell and Lisp.

The von Neumann architecture

The von Neumann architecture, named after Hungarian-American mathematician and computer scientist John von Neumann, is the most widely used architecture for building digital computers. The von Neumann architecture consists of a central processing unit (CPU), memory, and input/output (I/O) devices. The CPU fetches instructions from memory, decodes them, and executes them, and stores the results back in memory. The von Neumann architecture is based on the concept of stored-program computing, where instructions and data are stored in the same memory and can be accessed by the CPU in a uniform way. The von Neumann architecture is a sequential model of computation, meaning that instructions are executed one at a time in a specific order. This makes it well-suited for tasks that require a lot of sequential processing, such as scientific simulations and data analysis. However, it can be less efficient for tasks that can be parallelized, such as video encoding or machine learning.

Parallel computing

Parallel computing is a model of computation that aims to take advantage of multiple processing units to perform computations more quickly. Parallel computing can be applied to a wide range of problems, including scientific simulations, data analysis, and machine learning. There are several different models of parallel computing, including shared-memory parallelism, message-passing parallelism, and data-parallelism [3].

Shared-memory parallelism refers to the use of multiple processing units that share a common memory space. This model is typically used in multicore CPUs, where each core can execute a different thread of a program. However, shared-memory parallelism can lead to issues with synchronization and race conditions, where multiple threads access the same memory location at the same time.

Message-passing parallelism refers to the use of multiple processing units that communicate with each other by sending messages. This model is typically used in distributed computing systems, where multiple computers are connected over a network [4]. In message-passing parallelism, each processing unit has its own memory, and communication between units is done explicitly through message passing. This model can be more scalable than shared-memory parallelism, but it can also be more complex to program.

Data-parallelism refers to the use of multiple processing units that work on different pieces of data simultaneously. This model is well-suited for problems that can be divided into independent sub-problems, such as matrix multiplication or image processing. In data-parallelism, each processing unit operates on a separate portion of the data, and the results are combined at the end. This model can be very efficient for certain types of problems, but it may require special hardware or programming models.

Other models of computation

In addition to the models of computation discussed above, there are several other models that have been developed over the years. These include:

a) **Cellular automata:** A cellular automaton is a grid of cells that can be in a finite number of states. The cells update their state based on the state of their neighbors, according to a set of rules. Cellular automata have been used to model a wide range of phenomena, including biological systems, physical systems, and social systems.

b) **Quantum computing:** Quantum computing is a model of computation that uses quantum-mechanical phenomena, such as superposition and entanglement, to perform computations. Quantum computing has the potential to solve certain problems much faster than classical computers, but it also poses several challenges, including the need for error correction and the difficulty of programming quantum algorithms.

c) **Neural networks:** A neural network is a computational model inspired by the structure and function of the human brain. A neural network consists of a set of interconnected nodes (or neurons) that perform simple computations on their inputs. Neural networks have been used to solve a wide range of problems, including image recognition, natural language processing, and game playing.

When creating asynchronous and concurrent programmes that regularly use bit-level operations, a concurrent model of computation and a language based on the model for such operations are helpful. Video game software, hardware emulation (including virtual machines), and signal processing are a few examples. Few models and languages, though, are developed with bit-level concurrent processing in mind. For bit-level concurrent programming, we previously created the visual programming language A-BITS. The language uses processes that enable serial bit-level operations and FIFO buffers attached to them to compute using a concept akin to dataflow. Bit-level computing can be expressed naturally and developed in compositional ways. Next, for bit-level concurrent computation, we developed the APEC (Asynchronous Program Elements Connection) concurrent computation model. With the help of this model, the computation process can be defined precisely and formally, and the idea of basic programme components for operating and controlling may be expressed synthetically [5].

The difficulties in modelling cyber-physical systems (CPSs) that result from the systems' inherent heterogeneity, parallelism, and temporal sensitivity. The issues are shown using a section of an aircraft vehicle management system (VMS), specifically the fuel management subsystem, and then solutions that at least partially address the challenges are discussed. The specific technologies covered include heterogeneous and concurrent models of computing, domain-specific ontologies used to increase modularity, and combined modelling of functionality and implementation architectures [6]. Many distributed systems require atomicity, consistency, isolation, and durability as fundamental characteristics. The ACID characteristics are a common acronym for them. The cost of maintaining ACID is that the atomic actions must be completed flawlessly or they must be rolled back, which necessitates additional computation and network resources. Higher performance demands force us to either completely forgo the ACID qualities or compromise for eventual consistency. Such algorithms can become highly complex since they must account for all potential contingencies due to the ambiguity of the order of the occurrences. The Traquest model makes an effort to develop an universal concurrency paradigm that can deliver the ACID features without substantially degrading speed[7], [8].

Three different sets of experiments are used to test and compare a thorough three-dimensional concurrent flame spread model. Gravity, flow rate, pressure, oxygen mole percentage, and sample breadth are only a few of the variables that were changed. The calculated steady spread rate and flame profiles for buoyant flows (normal and partial gravities) are in good agreement with experiment. Although lower than expected, the extinction limits can be raised. The forced concurrent flow comparison in microgravity reveals the right trends. If the flames are short, the anticipated steady spread rates are lower than the experimental ones, and if the flames are long, they are greater. At the conclusion of the 5-s microgravity drops, it is thought that the experimental flames may not have quite reached steady state. Further longer-term microgravity tests will be required to support this theory [9].

DISCUSSION

Concurrent models of computation refer to computational models that allow multiple processes or threads to execute simultaneously. The aim is to take advantage of the parallelism inherent in modern computer systems to enhance the performance of computing tasks. Concurrent models are essential for developing high-performance computing systems, including those used for scientific simulations, graphics rendering, and data analysis.

There are various models of concurrent computation, each with its unique features, strengths, and limitations. Some of the most popular models include the message-passing model, shared-memory model, dataflow model, and process calculi. This article provides an overview of these models, highlighting their characteristics, advantages, and applications.

Message-Passing Model

The message-passing model is a form of concurrent computing that emphasizes the communication between processes. In this model, each process has its memory space, and communication between processes is achieved through message passing. The communication takes place through a set of primitives that allow the processes to send and receive messages to and from each other [10]. The message-passing model has several advantages, including its simplicity, ease of use, and support for distributed computing. It is also highly scalable, making it suitable for parallel computing tasks that involve a large number of processors. The message-passing model is widely used in high-performance computing and cluster computing systems.

Shared-Memory Model

The shared-memory model is a concurrent computing model that allows multiple processes to access the same memory space simultaneously. In this model, processes communicate by reading and writing to shared variables. The model has several advantages, including its simplicity, ease of use, and support for the use of standard programming languages.

One of the key challenges of the shared-memory model is ensuring that the different processes access shared memory in a coordinated manner to avoid race conditions and other synchronization problems. To overcome this challenge, various synchronization mechanisms have been developed, including locks, semaphores, and monitors. The shared-memory model is widely used in parallel computing applications, including scientific simulations, data analytics, and web servers.

Dataflow Model

The dataflow model is a concurrent computing model that emphasizes the flow of data between processes. In this model, processes communicate by exchanging data tokens, with each token representing a unit of data. The flow of data through the system is determined by the dependencies between the processes. The dataflow model has several advantages, including its ability to exploit parallelism automatically and its support for dynamic task scheduling. The model is also highly scalable and can handle large-scale parallel computing tasks with thousands of processors. The dataflow model is widely used in scientific computing, data analytics, and machine learning applications.

Process Calculi

Process calculi is a family of formal languages for describing concurrent systems. These languages are based on mathematical models of computation and provide a way to reason about the behavior of concurrent systems rigorously. Process calculi are used to develop concurrent systems, verify their correctness, and analyze their performance. Figure 1 illustrate computational model.

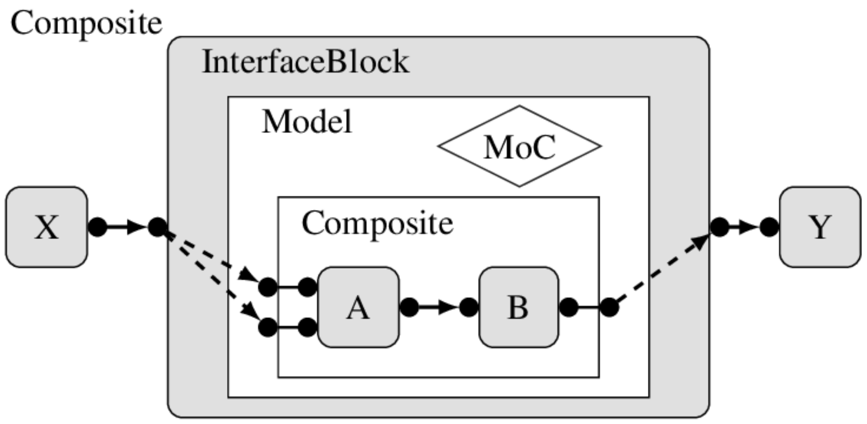


Figure 1:Illustrate the computational model.

One of the most popular process calculi is the pi calculus, which was developed by Robin Milner in the 1990s. The pi calculus provides a formalism for describing the behavior of concurrent systems based on the exchange of messages between processes. It has been used to develop many concurrent systems, including network protocols, web services, and distributed applications.

Another popular process calculus is the join calculus, which was developed by Cedric Fournet and Georges Gonthier. The join calculus provides a formalism for describing the behavior of systems that communicate through shared data structures. It has been used to develop many concurrent systems, including the operating system for Microsoft's Singularity research project.

Concurrent models of computation are essential for developing high-performance computing systems that can exploit the parallelism inherent in modern computer systems. There are various models of concurrent computation, each with its unique features, strengths, and limitations.

Some of the key factors that determine the choice of a particular model include the nature of the application, the hardware architecture of the system, and the performance requirements of the system. In recent years, the rise of multicore processors and the increasing popularity of distributed computing have made concurrent models of computation even more relevant. As the demand for high-performance computing continues to grow, it is likely that concurrent models of computation will play an even more significant role in shaping the future of computing.

In addition to the models discussed in this article, there are several other models of concurrent computation, including the actor model, the tuple space model, and the Linda model. These models have their unique features and applications, and researchers continue to explore their potential for developing high-performance computing systems concurrent models of computation are a fascinating area of computer science that holds great promise for improving the performance of computing systems. As computer hardware continues to evolve, it is likely that new models of concurrent computation will emerge, leading to even more powerful and efficient computing systems.

In good engineering practice, systems are constructed from a collection of parts. To fully comprehend the composition, we must first fully comprehend each of its constituent parts, and only then can we fully comprehend the significance of how those parts interact with one another well-conceived. With this composition, the components are clearly stated (as they are FSMs), but there are numerous interpretations that may be made for how the components interact. Semantics refers to a composition's intended meaning.

The semantics of concurrent composition are the main topic of this chapter. The literal meaning of the word "concurrent" is "running together." A system is considered concurrent if various system components theoretically execute simultaneously. These activities are conducted in any order. Nonetheless, the semantics of such concurrent operation can be very nuanced. In this chapter, we focus on actors, which respond to input port stimuli and generate output port stimuli. We shall only be tangentially interested in the definition of the actors themselves. These could be programmes written in an imperative programming language, hardware, or FSMs. The actions that these actors can take will need to be limited, but their specification does not need to be.

Three sets of rules, which we refer to as a model of computation, regulate the semantics of a concurrent composition of actors (MoC). The definition of a component is provided in the first set of rules. The concurrency mechanisms are described in the second set. The third describes the channels of communication.An actor with ports and a list of execution actions will serve as the component in this work.An execution action describes the actor's response to inputs in order to generate outputs and alter state. The ports will be connected to allow for communication between actors, and the environment of the actor will call the execution actions to make the actor carry out its duties. For FSMs, one action that results in a reaction is given. This chapter's main goal is to introduce a few of the potential concurrency and communication techniques that could control how these actors interact with one another.

We start by outlining the general model structure that all MoCs examined in this chapter share. The next step is to characterise a group of MoCs. Any model structure can be used with the aforementioned technique. It is not really necessary to change it to the format. As an alternative, we can simply start by labelling all signals as unknown and then, in any sequence, look at each actor to see what can be inferred about the outputs from its initial state. We keep doing this until there is no more progress possible, at which point either all signals are known or the model is rejected as being either flawed or non-constructive. Once all signals are known, all actors are able to change states, and we repeat the process in the new state for the subsequent reaction.

The aforementioned constructive method can be modified to handle nondeterministic machines. But now, things get much more nuanced, and there are other semantic nuances. Making an arbitrary option when using the constructive technique and coming into a nondeterministic choice is one way to deal with nondeterminism. We can either reject the model (not all choices result in well-formed or constructive models) or reject the decision and try again if the outcome causes the technique to fail to locate a fixed point.

Actors respond simultaneously and instantly under the SR model of computation, at least conceptually. Tight computing coordination is necessary to accomplish this with realistic computation. Next, we look at a group of computation models that require less cooperation. We take into account MoCs that are considerably more asynchronous than SR. Responses could happen at once or they might not. It is not a necessary component of the semantics whether they do or do not. The choice of when a reaction takes place might be made much more decentralizedly, possibly even by each individual actor. Reactions that are interdependent on one another are so because of data flow, not because events happened at the same time. If an actor a reaction needs information that was generated by an actor B reaction, then the A reaction must follow the B reaction. A dataflow model of computing is one where such data dependencies are the main limitations on reactions. Dataflow MoCs come in a variety of forms, some of which we take into consideration here. We will generally need scheduling policies that supply finite buffers for dataflow models that can execute with no bounds.

Deadlock is a potential second issue that might occur. Deadlock happens when a directed loop has insufficient tokens to satiate any of the firing rules of the loop's actors and there are cycles. Because to its capacity to generate an initial output token even in the absence of any available input tokens, the Wait actor can assist in avoiding stalemate. Delay actors or an equivalent are typically required in every cycle for dataflow models with feedback. It can be challenging to predict whether generic dataflow models will deadlock and whether there is an unlimited execution with bounded buffers. In fact, none of these two problems can be resolved by any method in bounded time for all dataflow models, making them undecidable. Fortunately, there are useful restrictions we can put on actor design that allow us to decide these issues. Next, we look at those limitations. A SDF model is said to be consistent if the balancing equations have a non-zero solution. It is contradictory if there is just one possible outcome: zero. With bounded buffers, an inconsistent model lacks unbounded execution.

According to Lee and Messerschmitt (1987), if the balancing equations have a solution that is not zero, they also have a solution in which qi is a positive integer for each actor i. Also, they provided a method for locating the least positive integer answer for connected models where there is a communication path between any two players. A scheduler for SDF models is built using such a process. Bounded buffers can be ensured by consistency, but an unbounded execution cannot be guaranteed by consistency. Deadlock may happen, particularly when there is feedback.

The SDF model gives Delay actors additional treatment to accommodate input. The Delay actor can create output tokens prior to receiving any input tokens, after which it acts as a simple SDF actor by copying inputs to outputs. Yet, the requirement for such a pointless actor is overstated, as is the expense of replicating inputs to outputs. It is quite efficient to use the Delay actor as a connection with initial tokens those tokens that the actor is able to produce before receiving inputs. The initial tokens must be taken into consideration by the scheduler. The ability to provide bounded buffers and prevent deadlock is valuable, but it has a cost. SDF doesn't express itself well. For instance, conditional firing, where an actor only fires if, say, a token has a certain value, cannot be expressed simply.

Dynamic dataflow, a more general dataflow model on which conditional firing is supported (DDF). DDF actors, in contrast to SDF actors, are not confined to producing the same number of output tokens with each firing and are allowed to have various firing rules. Without the requirement for specific handling of starting tokens, the DDF MoC immediately supports the Delay actor of two fundamental actors called Switch and Select are also mentioned.

There are three firing rules for the Select actor on the left. It needs one token on the bottom input port to start. Because that port has a Boolean type, its value must either be true or false. The actor does not output anything if a token with the value true is received on that input port; instead, it triggers the subsequent firing rule, which needs a token for the top-left input port with the letter T. The token on the T port is consumed by the actor's subsequent firing, which transfers it to the output port. The actor initiates a firing rule that needs a token on the bottom left input port with the letter F if a token with the value false is received on the bottom input port. After consuming that token, it delivers it once more to the output port. In order to produce one output, it fires twice.

In addition, the Switch actor serves another purpose. It simply has one firing rule, and that rule calls for a single token on each of the input ports. Depending on the Boolean value of the token received on the bottom input port, the token on the left input port will be delivered to either the T or the F output port.

CONCLUSION

Concurrent models of computation play a vital role in modern computing systems that require high-performance and scalability. These models enable multiple processes or threads to execute simultaneously, taking advantage of the parallelism inherent in modern computer systems. The message-passing model, shared-memory model, dataflow model, and process calculi are some of the most popular models of concurrent computation. Each of these models has its unique features, strengths, and limitations, and the choice of a particular model depends on the nature of the application, the hardware architecture of the system, and the performance requirements of the system.

REFERENCES:

[1] M. Wolczko, “Actors: A model of concurrent computation in distributed systems,” *Sci. Comput. Program.*, 1988, doi: 10.1016/0167-6423(88)90028-7.

[2] E. Lee and S. Neuendorffer, “Concurrent models of computation for embedded software,” in *System-on-Chip: Next Generation Electronics*, 2006. doi: 10.1049/PBCS018E_ch7.

[3] J. Kowalik, “ACTORS: A Model of Concurrent Computation in Distributed Systems (Gul Agha),” *SIAM Rev.*, 1988, doi: 10.1137/1030027.

[4] J. Xue, Z. Yang, S. Hou, and Y. Dai, “Processing Concurrent Graph Analytics with Decoupled Computation Model,” *IEEE Trans. Comput.*, 2017, doi: 10.1109/TC.2016.2618923.

[5] A. Takashi and T. Kensei, “A model of computation for bit-level concurrent computing and programming: APEC,” *IEICE Trans. Inf. Syst.*, 2008, doi: 10.1093/ietisy/e91-d.1.1.

[6] P. Derler, E. A. Lee, and A. Sangiovanni Vincentelli, “Modeling cyber-physical systems,” *Proc. IEEE*, 2012, doi: 10.1109/JPROC.2011.2160929.

[7] H. Azeem, S. Yellasiri, V. Jammala, B. S. Naik, and A. K. Panda, “A Fuzzy Logic Based Switching Methodology for a Cascaded H-Bridge Multi-Level Inverter,” *IEEE Trans. Power Electron.*, 2019, doi: 10.1109/TPEL.2019.2907226.

[8] K. Chen, Q. Deng, Y. Hou, Y. Jin, and X. Guo, “Hardware and software co-verification from security perspective,” in *Proceedings - 2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification, MTV 2019*, 2019. doi: 10.1109/MTV48867.2019.00018.

[9] S. Y. Hsu and J. S. T’ien, “Flame spread over solids in buoyant and forced concurrent flows: Model computations and comparison with experiments,” *Proc. Combust. Inst.*, 2011, doi: 10.1016/j.proci.2010.05.093.

[10] H. Gao, H. Yu, G. Xie, H. Ma, Y. Xu, and D. Li, “Hardware and software architecture of intelligent vehicles and road verification in typical traffic scenarios,” *IET Intell. Transp. Syst.*, 2019, doi: 10.1049/iet-its.2018.5351.

CHAPTER 9

AN EXTENSIVE STUDY ON SENSORS AND ACTUATORS

Ms. Maitrayee Konar, Assistant Professor
Department of Electronics and Communication Engineering, Presidency University, Bangalore, India
Email id- maitraiyeekonar@presidencyuniversity.in

ABSTRACT:

Sensors and actuators are fundamental components in many engineering systems, providing a means of measuring and controlling physical quantities. Sensors are devices that detect and measure changes in their environment, while actuators are devices that produce a physical response or control action based on a signal from a controller. Together, sensors and actuators form a closed-loop control system that can be used to monitor and regulate various physical processes. Actuators, on the other hand, can be designed to produce a variety of mechanical responses, such as movement, vibration, or pressure. Some common types of actuators include electric motors, solenoids, and hydraulic cylinders. These devices are often controlled by a feedback loop that uses information from sensors to adjust the actuator output and achieve a desired result.

KEYWORDS:

Actuators, Devices, Environment, Hydraulic Cylinders, Solenoids.

INTRODUCTION

Sensors and actuators are essential components of modern technology. They play a crucial role in the functioning of various devices, systems, and machines. Sensors are used to detect changes in the environment, while actuators are used to produce a response based on the sensor data. These components are widely used in various industries, including manufacturing, automotive, healthcare, and aerospace. This article will discuss sensors and actuators in detail, including their types, applications, and how they work [1].A sensor is a device that detects changes in the environment and converts them into electrical signals. These signals are then processed by a microcontroller or a computer to determine the appropriate response. Sensors are used in various applications, including temperature measurement, pressure measurement, distance measurement, and motion detection.

Types of Sensors:

There are several types of sensors, including:

1. Temperature Sensors: Temperature sensors are used to measure the temperature of the environment. There are several types of temperature sensors, including thermocouples, thermistors, and RTDs.
2. Pressure Sensors: Pressure sensors are used to measure the pressure of the environment. There are several types of pressure sensors, including piezoelectric sensors, capacitive sensors, and strain gauge sensors.

3. Proximity Sensors: Proximity sensors are used to detect the presence of objects in the environment. There are several types of proximity sensors, including inductive sensors, capacitive sensors, and ultrasonic sensors.
4. Motion Sensors: Motion sensors are used to detect motion in the environment. There are several types of motion sensors, including accelerometers, gyroscopes, and magnetometers.
5. Light Sensors: Light sensors are used to measure the amount of light in the environment. There are several types of light sensors, including photodiodes, phototransistors, and photovoltaic cells.

Sensors work by detecting changes in the environment and converting them into electrical signals. The specific working mechanism of a sensor depends on its type. For example, a temperature sensor works by measuring the electrical resistance of a material, which changes as the temperature of the material changes. A pressure sensor works by measuring the amount of deformation in a material due to the pressure of the environment [2].An actuator is a device that produces a mechanical response based on the sensor data. Actuators are used in various applications, including robotics, manufacturing, and aerospace. Actuators are used to move, control, or manipulate objects in the environment.

Types of Actuators:

There are several types of actuators, including:

1. **Electric Actuators:** Electric actuators use an electric motor to produce mechanical motion. There are several types of electric actuators, including linear actuators, rotary actuators, and stepper motors.
2. **Pneumatic Actuators:** Pneumatic actuators use compressed air to produce mechanical motion. There are several types of pneumatic actuators, including cylinders and rotary actuators.
3. **Hydraulic Actuators:** Hydraulic actuators use a hydraulic fluid to produce mechanical motion. There are several types of hydraulic actuators, including cylinders and rotary actuators.
4. **Piezoelectric Actuators:** Piezoelectric actuators use the piezoelectric effect to produce mechanical motion. They are commonly used in micro- and nano-scale applications.

Actuators work by producing a mechanical response based on the sensor data. The specific working mechanism of an actuator depends on its type [3]. For example, an electric linear actuator works by converting the rotational motion of an electric motor into linear motion. A pneumatic cylinder works by using compressed air to move a piston inside a cylinder.

Applications of Sensors and Actuators:

1. **Manufacturing:** Sensors and actuators are widely used in manufacturing applications, including robotics and automation. They are used to detect changes in the environment and control the motion of machines and equipment.

2. **Automotive:** Sensors and actuators are used in automotive applications, including engine control, braking systems, and fuel injection systems. They are used to detect changes in the environment and adjust the performance of the vehicle accordingly.
3. **Healthcare:** Sensors and actuators are used in healthcare applications, including medical devices and diagnostic equipment. They are used to detect changes in the environment, such as body temperature and blood pressure, and produce a response, such as administering medication or adjusting medical equipment.
4. **Aerospace:** Sensors and actuators are used in aerospace applications, including flight control systems and satellite communication systems. They are used to detect changes in the environment and adjust the performance of the aircraft or satellite accordingly.
5. **Home Automation:** Sensors and actuators are used in home automation applications, including smart thermostats, lighting control systems, and security systems. They are used to detect changes in the environment, such as temperature and motion, and produce a response, such as adjusting the thermostat or turning on the lights.

Sensors and actuators are essential components of modern technology. They are used in various applications, including manufacturing, automotive, healthcare, aerospace, and home automation. Sensors are used to detect changes in the environment, while actuators are used to produce a response based on the sensor data. There are several types of sensors and actuators, each with a unique working mechanism and specific applications. Understanding the functioning of sensors and actuators is crucial for the development of modern technology and the advancement of various industries.

Electric parameters must be matched with the driver, source, cable, or receiver electronics in any electric transmission line involving the transfer of power or an electric signal. For piezoelectric sensors, actuators, and transducers, designing an electric impedance matching circuit requires careful consideration of the operating frequencies, transmitter or receiver impedance, power supply or driver impedance, and the impedance of the receiver electronics. The methods for matching the electric impedance of piezoelectric sensors, actuators, and transducers with their accessories, such as amplifiers, cables, power supplies, receiver electronics, and power storage, are discussed in this work. It has been discussed how to build a power supply, preamplifier, cable, and matching circuits for balancing electric impedance between sensors, actuators, and transducers. The common techniques, models, and material attributes used for the design of electric impedance matching are discussed at the outset of the work [4].

Many scientists are very interested in the creation of lightweight, stronger, and more flexible structures. Due to their intrinsic electromechanical interaction, piezoelectric materials have been frequently used in the fabrication of such structures to dampen vibrations. Nonetheless, determining the ideal control and sensor-actuator arrangement is one of the major issues. The benchmark model used in this study for the active vibration management for flexible structures is a plate that is only loosely supported. It uses a feedback controller with a sensor-actuator setup that is collocated. Using piezoelectric (PZT) patches, the plate is subjected to both a control and disturbance signal. The Euler-Bernoulli model serves as the foundation for the analytical model. Ant Colony Optimization (ACO) is used to establish the optimal location for the collocated sensor-actuator and PID controller gains, which are then compared to the Genetic Algorithm (GA) and enumerative method (EM).

Wireless sensor/actuator networks are used in many crucial and significant applications (WSAN). These applications have a wide spectrum and can be used in industrial, commercial, and residential settings. Most of these applications require reliable data delivery services in addition to some degree of real-time communication. A well-known standard for supporting publish/subscribe-based real-time distributed systems is data distribution service (DDS). The best-effort and fully-reliable quality-of-service levels of data reliability are provided by the DDS specification. For sensor-based platforms, TinyDDS is a lightweight and incomplete implementation of the DDS middleware designed especially for platforms with constrained resources. But, TinyDDS does not currently have the data delivery dependability support [5].

A new generation of sensor networks called wireless sensor/actuator networks (WSANs) is starting to emerge. WSANs, acting as the foundation of control applications, will make it possible for dispersed and mobile control to a completely new level. WSAN architecture, however, faces significant difficulties because to the unreliability of wireless communications and the real-time demands of control applications. This work proposes an application-level design technique for WSANs in mobile control applications with a focus on the reliability issue. Insofar as it is unaffected by the underlying platforms, surroundings, control system concepts, and controller architecture, the solution is general. Experiments are run on a genuine WSAN system to capture the link quality features in terms of packet loss rate. An easy-to-use yet effective solution is suggested to deal with unpredictable packet loss on actuator nodes as a result of the experimental observations[6].

The key challenge in high-dimensional estimation and control is the selection of the best sensors and actuators. These sensor and actuator placements influence almost all future control decisions. In this article, we use greedy optimization and balanced model reduction to effectively choose sensors and actuators that maximise observability and controllability. With the help of a greedy matrix QR that is centred on the dominant modes of the direct and adjoint balancing transformations, we specifically identify the regions that optimise scalar measures of observability and controllability. Because the pivoting runtime increases linearly with the state size, this approach can be used to systems with high dimensions. The linearized Ginzburg-Landau system is used to demonstrate the results. For this system, our algorithm comes close to the known optimal placements that were calculated using pricey gradient descent techniques.

It is now possible to organise new materials at the nanoscale thanks to breakthroughs in nanotechnology, which opens up the possibility of creating unique material systems and gadgets that are capable of self-awareness and active response. Carbon nanotubes are excellent prospects for future multi-functional material systems that integrate adaptive and sensing capabilities because of the intrinsic link between their electrical characteristics and mechanical deformation. A fundamental understanding of these materials' structure/property relationships is required for the development of these material systems with multifunctional components for sensing and actuation.

The performance of industrial cyber-physical systems can be negatively impacted by controller failures, which can also lead to unsafe physical plant operations. In order to increase the resistance of control systems to issues and potential physical failures, we describe in this paper a controller switching method over wireless sensor-actuator networks. To maintain the stability of the control system in the event that the primary controller fails, the suggested mechanism promptly identifies controller faults and switches to the backup controller. We undertake a performance evaluation on a hardware-in-the-loop testbed that takes into account both the real wireless network protocol and the simulated physical system in order to demonstrate the effectiveness of our suggested approach. Findings show that, in the event of a controller failure, the suggested scheme swiftly recovers by switching to a backup controller[7], [8].

Carlos Gonclaves et al. Wearable e-textiles are considered a means to add features to standard wearable textiles and provide competitive market advantages since they can carry out electronic operations. The creation of e-textiles is now both a research project and a production problem for industry. It's critical to understand how to leverage current industrial processes or to create new ones that can scale up production while maintaining the functionality and performance of prototypes. While there are significant technological difficulties, there are already several instances of wearable e-textiles where sensors, actuators, and fabrication processes were utilised to seamlessly incorporate electronic components into conventional wearable textiles, allowing for everyday usage without a bionic stigma [9].

DISCUSSION

Sensors and actuators are fundamental components of modern-day technology. They play an important role in a wide range of applications, from automotive and aerospace systems to consumer electronics and medical devices. In this essay, we will provide an introduction to sensors and actuators, their definitions, types, applications, and their role in various industries.

1. A sensor is a device that detects and measures physical or chemical properties and converts them into a readable signal. An actuator, on the other hand, is a device that takes input from a sensor and produces a mechanical or electrical output. Sensors and actuators work together to perform a specific task, such as monitoring temperature or controlling a robotic arm.

2. There are many different types of sensors, each designed to detect specific physical or chemical properties. Some of the most common types of sensors include:

   i. Temperature Sensors Temperature sensors are used to measure the temperature of an object or environment. There are several types of temperature sensors, including thermocouples, RTDs, and thermistors.
   ii. Pressure Sensors Pressure sensors are used to measure the pressure of a gas or liquid. They can be used in a variety of applications, including automotive systems and medical devices.
   iii. Motion Sensors Motion sensors are used to detect movement. They are commonly used in security systems, gaming devices, and smartphones.
   iv. Light Sensors Light sensors are used to detect the presence or absence of light. They can be used in automatic lighting systems and cameras.
   v. Chemical Sensors Chemical sensors are used to detect specific chemicals or chemical properties. They are commonly used in environmental monitoring and medical devices.

3. Types of Actuators Actuators are devices that convert energy into mechanical or electrical motion. Some of the most common types of actuators include:

   i. Electric Actuators Electric actuators use electricity to produce mechanical motion. They are commonly used in industrial automation and robotics.

ii. Hydraulic Actuators Hydraulic actuators use a liquid, such as oil or water, to produce mechanical motion. They are commonly used in heavy machinery and automotive systems.
iii. Pneumatic Actuators Pneumatic actuators use compressed air to produce mechanical motion. They are commonly used in automotive systems and industrial automation.
iv. Piezoelectric Actuators Piezoelectric actuators use an electric field to produce mechanical motion. They are commonly used in medical devices and precision positioning systems.

4. Applications of Sensors and Actuators Sensors and actuators are used in a wide range of applications, including:

i. Automotive Systems Sensors and actuators are used in a variety of automotive systems, including engine management, airbag deployment, and antilock braking systems.
ii. Aerospace Systems Sensors and actuators are used in aerospace systems, including flight control systems, navigation systems, and engine management.
iii. Medical Devices Sensors and actuators are used in medical devices, including insulin pumps, pacemakers, and diagnostic equipment.
iv. Consumer Electronics Sensors and actuators are used in a variety of consumer electronics, including smartphones, gaming devices, and wearable technology.
v. Robotics Sensors and actuators are used in robotics, including industrial automation and consumer robotics.

5. Conclusion Sensors and actuators are essential components of modern-day technology. They play a critical role in a wide range of applications, from automotive and aerospace systems to consumer electronics and medical devices. As technology continues to advance, the use of sensors and actuators is expected to grow, creating new opportunities for innovation and advancement.

6. Role of Sensors and Actuators in Various Industries

i. The automotive industry relies heavily on sensors and actuators to ensure the safety and reliability of vehicles. Sensors are used to monitor engine performance, tire pressure, and vehicle speed, while actuators are used to control braking and steering systems. In addition, sensors and actuators are used in the development of autonomous vehicles, which rely on a complex network of sensors and actuators to navigate and make decisions on the road[10].
ii. The aerospace industry also relies heavily on sensors and actuators to ensure the safety and reliability of aircraft. Sensors are used to monitor engine performance, fuel levels, and air pressure, while actuators are used to control flight surfaces and engine thrust. In addition, sensors and actuators are used in the development of unmanned aerial vehicles (UAVs), which rely on sensors and actuators to navigate and perform tasks.
iii. The medical industry uses sensors and actuators in a variety of devices, including diagnostic equipment, pacemakers, and insulin pumps. Sensors are used to monitor vital signs, blood glucose levels, and other physiological parameters, while actuators are used to control drug delivery and other therapeutic interventions. In addition, sensors and actuators are used in the development of prosthetics and exoskeletons, which rely on sensors and actuators to provide mobility and function to individuals with physical disabilities.

iv. Consumer Electronics Industry The consumer electronics industry relies heavily on sensors and actuators to provide enhanced functionality and user experience. Sensors are used in smartphones to provide touch screen functionality and motion detection, while actuators are used to provide haptic feedback and other tactile responses. In addition, sensors and actuators are used in gaming devices and virtual reality systems to provide immersive and interactive experiences.

v. The robotics industry relies heavily on sensors and actuators to provide mobility and functionality to robotic systems. Sensors are used to provide information on the environment and enable robotic systems to navigate and perform tasks, while actuators are used to control the movement and operation of robotic systems. In addition, sensors and actuators are used in the development of human-robot interaction systems, which rely on sensors and actuators to provide feedback and respond to human actions and commands.

7. Challenges and Future Directions

Despite the significant progress made in the development and implementation of sensors and actuators, there are still challenges that need to be addressed. One of the main challenges is the integration of sensors and actuators into complex systems and networks, which requires the development of advanced algorithms and control systems. In addition, there are challenges related to power consumption, size, and cost, which can limit the performance and scalability of sensor and actuator systems.

The future of sensors and actuators is promising, with continued advancements in technology and the development of new applications and use cases. Some of the key areas of focus include the development of flexible and stretchable sensors, which can be integrated into wearable technology and medical devices, as well as the development of advanced sensor and actuator networks for autonomous systems and smart cities. In addition, there is significant potential for the development of novel sensors and actuators based on emerging technologies, such as nanotechnology and biotechnology. In Figure 1 illustrate the actuator flow.

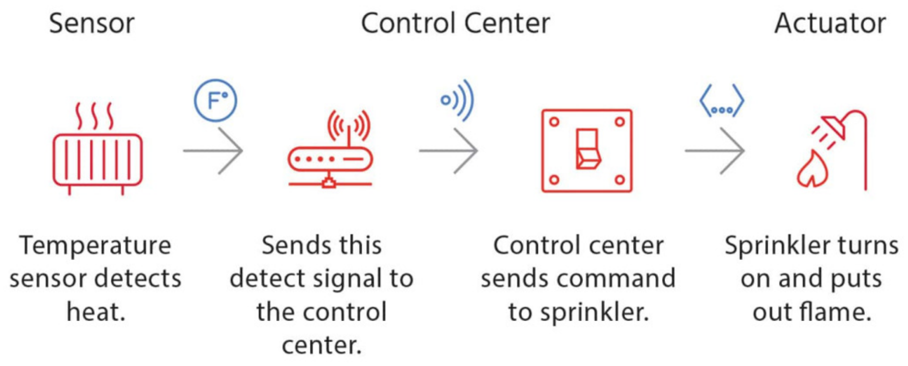

Sensor
Control Center
Actuator
F°
Temperature sensor detects heat.
Sends this detect signal to the control center.
Control center sends command to sprinkler.
Sprinkler turns on and puts out flame.

Figure 1: Illustrate the sensor to actuator flow.

The development and use of sensors and actuators raise ethical considerations related to privacy, data security, and the potential for misuse. The widespread use of sensors in smart cities and other public spaces raises concerns about the collection and use of personal data, as well as the potential for surveillance and invasion of privacy. In addition, the use of sensors in medical devices and other healthcare applications raises concerns about the accuracy and reliability of the data collected, as well as the potential for errors or malfunctions that could have serious consequences for patient health and safety.

CONCLUSION

Sensors and actuators are essential components of modern technology and have revolutionized the way we interact with and control the physical world. They enable the collection and processing of data from the environment, which is then used to make decisions, provide feedback, and control systems. Sensors and actuators have applications in various industries, including automotive, aerospace, medical, consumer electronics, and robotics, and are expected to continue to drive innovation and advancements in technology. The development and use of sensors and actuators raise ethical considerations related to privacy, data security, and the potential for misuse, and these concerns must be carefully considered and addressed to ensure that the benefits of this technology are realized while minimizing potential harms.

REFERENCES:

[1] D. Chen *et al.*, "4D Printing Strain Self-Sensing and Temperature Self-Sensing Integrated Sensor–Actuator with Bioinspired Gradient Gaps," *Adv. Sci.*, 2020, doi: 10.1002/advs.202000584.

[2] X. Jin *et al.*, "Review on exploration of graphene in the design and engineering of smart sensors, actuators and soft robotics," *Chemical Engineering Journal Advances*. 2020. doi: 10.1016/j.ceja.2020.100034.

[3] F. Xia, "QoS challenges and opportunities in wireless sensor/actuator networks," *Sensors*. 2008. doi: 10.3390/s8021099.

[4] V. T. Rathod, "A review of electric impedance matching techniques for piezoelectric sensors, actuators and transducers," *Electronics (Switzerland)*. 2019. doi: 10.3390/electronics8020169.

[5] A. A. Al-Roubaiey, T. R. Sheltami, A. S. H. Mahmoud, and K. Salah, "Reliable Middleware for Wireless Sensor-Actuator Networks," *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2893623.

[6] F. Xia, Y. C. Tian, Y. Li, and Y. Sun, "Wireless sensor/actuator network design for mobile control applications," *Sensors*, 2007, doi: 10.3390/s7102157.

[7] H. Azeem, S. Yellasiri, V. Jammala, B. S. Naik, and A. K. Panda, "A Fuzzy Logic Based Switching Methodology for a Cascaded H-Bridge Multi-Level Inverter," *IEEE Trans. Power Electron.*, 2019, doi: 10.1109/TPEL.2019.2907226.

[8] K. Chen, Q. Deng, Y. Hou, Y. Jin, and X. Guo, "Hardware and software co-verification from security perspective," in *Proceedings - 2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification, MTV 2019*, 2019. doi: 10.1109/MTV48867.2019.00018.

[9] C. Gonçalves, A. F. da Silva, J. Gomes, and R. Simoes, "Wearable e-textile technologies: A review on sensors, actuators and control elements," *Inventions*, 2018, doi: 10.3390/inventions3010014.

[10] S. Raikwar, L. Jijyabhau Wani, S. Arun Kumar, and M. Sreenivasulu Rao, “Hardware-in-the-Loop test automation of embedded systems for agricultural tractors,” *Meas. J. Int. Meas. Confed.*, 2019, doi: 10.1016/j.measurement.2018.10.014.

CHAPTER 10

EMBEDDED SENSOR BASED SYSTEMS

Dr. Rajbhadur Singh, Assistant Professor
Department of Computer Science and Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- rajbhadurs.soeit@sanskriti.edu.in

ABSTRACT:

Embedded processors are specialized computer chips designed for use in electronic systems and devices, such as cars, smartphones, and medical equipment. These processors are designed to perform specific functions and are integrated into the hardware of the system, making them an essential part of the technology that we rely on every day. Embedded processors have advanced capabilities, including the ability to process large amounts of data, communicate with other devices, and control complex systems. They are designed to be energy-efficient and have low power consumption, making them ideal for use in battery-powered devices.

KEYWORDS:

Computer Chips, Complex System, Embedded Processors, Data Communication, Hardware.

INTRODUCTION

An embedded processor is a microprocessor that is designed to be integrated into a system, rather than used as a standalone computer. These processors are commonly used in a wide range of applications, including industrial control, automotive electronics, and consumer electronics. They are designed to perform specific functions, and they are typically optimized for low power consumption, small size, and high reliability [1], [2]. This will provide an overview of embedded processors, including their features, applications, and design considerations.

Features of Embedded Processors:

Embedded processors have several features that distinguish them from general-purpose processors. Some of these features include:

1. **Real-time operation:** Embedded processors are designed to operate in real-time, meaning that they can respond quickly to changes in their environment. This is particularly important in applications such as industrial control and automotive electronics, where safety is critical.
2. **Low power consumption:** Embedded processors are typically designed to operate on low power, making them ideal for applications where power efficiency is critical.
3. **Small size:** Embedded processors are often designed to be compact and lightweight, making them ideal for use in small devices and systems.
4. **High reliability:** Embedded processors are typically designed to operate continuously without failure. This is particularly important in applications such as aerospace, where failure can have catastrophic consequences.

Applications of Embedded Processors:

Embedded processors are used in a wide range of applications, including:

1. **Industrial Control:** Embedded processors are used in industrial control systems, where they are used to monitor and control various processes. They are commonly used in applications such as manufacturing, oil and gas production, and building automation.
2. **Automotive Electronics:** Embedded processors are used in automotive electronics, where they are used to control various systems, such as the engine, transmission, and brakes. They are also used in driver assistance systems, such as collision avoidance and lane departure warning systems.
3. **Consumer Electronics:** Embedded processors are used in consumer electronics, including smartphones, digital cameras, and gaming consoles. They are used to control various functions of these devices, including processing images and video, managing storage, and connecting to wireless networks.
4. **Aerospace:** Embedded processors are used in aerospace applications, including satellites, avionics, and unmanned aerial vehicles. They are used to control various systems, including navigation, communication, and propulsion.

Design Considerations for Embedded Processors:

When designing a system with embedded processors, there are several factors that must be taken into consideration. Some of these factors include:

1. **Power Consumption:** Embedded processors must be designed to operate on low power, which can be challenging when they are required to perform complex tasks [3]. Power management techniques, such as clock gating and power gating, can be used to reduce power consumption.
2. **Memory:** Embedded processors typically have limited memory, which can limit their ability to perform complex tasks. Memory management techniques, such as caching and virtual memory, can be used to maximize the use of available memory.
3. **Real-Time Performance:** Embedded processors must be designed to operate in real-time, which can be challenging when they are required to perform complex tasks. Real-time operating systems (RTOS) and scheduling algorithms can be used to ensure that tasks are executed in a timely manner.
4. **Reliability:** Embedded processors must be designed to operate continuously without failure, which can be challenging when they are required to perform complex tasks. Techniques such as redundancy and error correction can be used to improve reliability.

Types of Embedded Processors:

There are several types of embedded processors, including:

1. **Microcontrollers:** Microcontrollers are a type of embedded processor that combines a microprocessor with memory and input/output (I/O) interfaces on a single chip. They are

commonly used in applications such as industrial control, automotive electronics, and consumer electronics.

2. **Digital Signal Processors (DSPs):** DSPs are a type of embedded processor that is designed to perform digital signal processing tasks. They are commonly used in applications such as audio and video processing, image processing, and telecommunications.
3. **Field Programmable Gate Arrays (FPGAs):** FPGAs are a type of embedded processor that can be programmed to perform specific functions. They are commonly used in applications such as aerospace, telecommunications, and industrial control.
4. **System-on-Chip (SoC):** SoC is a type of embedded processor that integrates several components, including a microprocessor, memory, and input/output (I/O) interfaces, onto a single chip. They are commonly used in applications such as smartphones, tablets, and other mobile devices.
5. **Graphics Processing Units (GPUs):** GPUs are a type of embedded processor that are designed to perform graphics and visual processing tasks. They are commonly used in applications such as gaming, virtual reality, and augmented reality.

High-performance embedded systems are needed to satisfy the demands of user-centered applications as a result of the growth of the mobile sector. Using compressed data is effective for an embedded system due to the memory resource limitations. The embedded CPU, however, has a severe bottleneck due to the effort associated with data decompression. By combining a hardware accelerator with the CPU to create a system-on-chip (SoC) for the embedded system, one of the bottlenecks may be reduced. In this article, we provide a lossless decompression accelerator for embedded processors that supports static Huffman decoding for an inflate method and LZ77 decompression. The accelerator is constructed using a Sam-sung 65 nm complementary metal oxide semiconductor (CMOS) technology and implemented on a field programmable gate array (FPGA) to test its functional suitability. The Canterbury corpus benchmark is used to assess the accelerator's performance, and throughput of up to 20.7 MB/s at 50 MHz system clock frequency was obtained.

Mingyu Yang et al. Via so-called eHealth systems, the Internet of Things' (IoT) rapid growth has created new potential for healthcare systems. Disease diagnoses may be made in real time with the aid of monitoring utilising portable IoT devices equipped with biomedical sensors. Nevertheless, there is a problem since IoT devices are battery-powered and have severe resource limitations, and monitoring is an always-on activity that needs a steady power supply. In this study, a low-power eHealth device is realistically implemented utilising both hardware and software techniques. We implement a number of memory-conscious dynamic temporal warping (DTW) algorithms to realise different lightweight eHealth applications, notably monitoring apps, for deployment on a tiny and low-power embedded CPU. The processor's actual prototypes are presently being created. This comparison shows how well our work performs in terms of circuit area (cost of manufacturing) and power efficiency when compared to other cutting-edge embedded processors. By altering the quantity of data utilised in DTW for different eHealth monitoring applications, we also showed the software implementation's potential to scale [4].

Sarah Azimi et al. High-performance microprocessors have been created as a result of the constant scaling of electronic components, and they are even suited for safety-critical applications where radiation-induced mistakes, such as single event effects (SEEs), are one of the most significant reliability challenges. The primary goal of this work is to create a fault injection environment that can analyse the effects of errors on the functionality of an ARM Cortex-A9 microprocessor embedded within a Zynq-7000 AP-SoC while taking into account various fault models that can affect the embedded processor's system memory and register resources. For the simulation of radiation-induced defects in the AP-SoC hardware resources during the execution of software applications, we created a unique Python-based fault injection platform. The fault injection technique is non-intrusive and doesn't call for changing the software programme that's being tested. On a portion of the MiBench benchmark software package, the experimental investigations were conducted. The effectiveness of the established approach and the capacity to assess different fault model sets are shown by the results of fault injection.

Mojtaba Ebrahimi et al. A significant problem in modern commercial electronic components and systems is radiation-induced soft mistakes. We now show the findings of an investigation of an embedded processor's soft error rate (SER). Starting with a technological response model generated using TCAD simulations at the device level and progressing through application masking, our SER analysis platform properly simulates generation, propagation, and masking effects. To offer the detailed contribution of each component (flip-flops, combinational gates, and SRAMs) to the overall SER, the platform combines precise models at the device level, analytical error propagation at the gate level, and fault emulation at the architecture/application level. The impact of mistakes is propagated up the modelling hierarchy at each step using the appropriate degree of abstraction. Analyzing the complete processor provides greater insight into the relative contributions of combinational and sequential SER than earlier research, which were based on extremely basic test circuits. The analysis's findings may help circuit designers use efficient hardening strategies to lower the total SER while still adhering to necessary power and performance limitations [5].

Xiang Wang et al. As more and more code tampering attacks and transient errors are substantially jeopardising the security of embedded processors, safe programme execution of embedded processors has drawn significant research interest. In addition to being intimately tied to embed device security, programme monitoring and fault recovery mechanisms also have a direct impact on processor performance. The two-stage checkpoint is regularly backed up using the security monitoring and fault recovery architecture presented in this work for run-time programme execution. In this framework, the programme execution is tracked in real-time using the integrity check technology based on the basic block (BB), and when the integrity check fails, the programme is rolled back. A Monitoring Cache (M-Cache) is also constructed to buffer the reference data for integrity verification. In order to guarantee the smooth operation of the embedded system, a recovery technique is offered, focusing primarily on three tampered places (registers in the processor, instructions in the cache, and codes in memory). The provided security architecture is finally implemented and verified using an open RISC processor, which has been shown to be efficient for detecting programmes during the execution of tamper attacks and for speedy recovery of the operating environment and code [6].

Seo, Hwajeong Seo et al. the first ARIA block cypher optimised implementation for 8-bit RISC microcontrollers by Alf and Vegard. Primitive operations, such as rotation, a substitution layer, and a diffusion layer, are carefully tuned for the target low-end embedded CPU in order to achieve high-speed implementation. The electronic codebook (ECB) and counter (CTR) modes of operation are supported by the planned ARIA implementation. In particular, the pre-computed table of two add-round-key, one substitution layer, and one diffusion layer operations further optimises the CTR mode of operation. Eventually, for the 128-bit, 192-bit, and 256-bit security levels, respectively, the suggested ARIA-CTR implementations on 8-bit AVR microcontrollers obtained 187.1, 216.8, and 246.6 clock cycles per byte. The execution time for the 128-bit, 192-bit, and 256-bit security levels, respectively, is improved by 69.8%, 69.6%, and 69.5% when compared to earlier reference implementations [7].

Stefanus Kurniawan et al. The Internet of Things (IoT) has additional uses that call for advanced processing while yet using little electricity. Embedded processors, which make up the heart of the Internet of Things system, use MPSoC, a parallel processing architecture, to address the problem of computing. While there is a vast range of MPSoC hardware available, there is only a limited amount of software support in the form of accessible libraries and development platforms. Such a platform is required to support concurrent embedded software research and development. The popular embedded development platform Arduino does not yet have formal multicore programming capability. The RUMPS401 low-power MPSoC is the focus of this work's proposal for an arduino-based development environment that allows multicore programming while keeping arduino's straightforward programme format. The suggested environment is completely functional and can be readily applied to additional MPSoCs with comparable topologies even if it only targets a single MPSoC [8].

DISCUSSION

Embedded processors often have a specific purpose when used in a product. They can manage an automobile engine or gauge Arctic ice thickness. They are not required to employ user-defined software to carry out arbitrary tasks. The processors may then be more specialised as a result. Increasing their specialisation may have a significant positive impact. For instance, they could use a lot less power and so can be used for extended periods of time with few batteries. Alternatively they could have specialist equipment to carry out tasks like image analysis that would be expensive to carry out on standard gear.

Understanding the distinction between an instruction set architecture (ISA), a processor realisation, and a chip is crucial when assessing CPUs. The latter is a silicon component that a semiconductor vendor is selling. The former is a description of the instructions that the processor can carry out as well as certain structural requirements that realisations must have, such word size. An ISA is x86. There are several insights. A common abstraction across numerous realisations is an ISA. A single ISA may be found in several chips, often built by various vendors and frequently with wildly differing performance characteristics.

Since software tools are expensive to design, it makes sense for a family of processors to share an ISA so that the same programmes may (sometimes) execute properly on different realisations. This latter trait, however, is rather dangerous since an ISA typically does not have any temporal restrictions. As a result, even if a programme may run theoretically the same on various processors, the system behaviour when the processor is incorporated in a cyber-physical system may be completely different.

A microcontroller (abbreviated as "C") is a miniature computer on a single integrated circuit that combines a central processing unit (CPU) with peripherals including memory, I/O devices, and timers. According to some estimates, microcontrollers make up more than half of all CPUs sold globally, however it is difficult to verify this claim since there is little distinction between microcontrollers and general-purpose processors. The simplest microcontrollers use 8-bit words and are best suited for applications that need less memory and straightforward logical operations (as opposed to performance-demanding arithmetic operations). They may only use a few nanowatts of power while sleeping, and they often have sleep modes that do the same. It has been shown that embedded components, including sensor network nodes and surveillance equipment, can run for many years on a little battery.

Microcontrollers may become quite complex. It may be challenging to distinguish them from general-purpose processors. For instance, the Intel Atom is a series of x86 Processors that is primarily utilised in netbooks and other compact mobile computers. These processors are suited for certain embedded applications and servers where cooling is a challenge since they are designed to use very little energy without significantly reducing performance in comparison to processors used in higher-end systems. Another example of a CPU that straddles the hazy line between general-purpose processors and microcontrollers is AMD's Geode.

DSP Processors

In many embedded applications, signal processing is used extensively. A signal is a series of physically collected data acquired at a regular pace known as the sample rate. With sample rates ranging from a few Hertz (Hz, or samples per second) to a few hundred Hertz, a motion control application, for instance, may read position or location data from sensors. From 8,000 Hz (or 8 kHz, the sample rate used in telephony for speech communications) to 44.1 kHz (the sample rate of CDs), audio signals are sampled. Sound signals may be sampled at even greater rates for ultrasonic applications (like medical imaging) and high-performance musical instruments. For consumer devices, video normally employs sample rates of 25 or 30 Hz, whereas significantly higher rates are used for specialised measuring purposes. Naturally, each sample comprises a whole picture (referred to as a frame), which is composed of several samples (referred to as pixels) that are spread in space rather than time. Applications for software-defined radio may use sample rates as high as several GHz (for baseband processing) (billions of Hertz). Interactive games, radar, sonar, and LIDAR imaging systems, video analytics (the extraction of information from video, for example for surveillance), driver-assist systems for cars, medical electronics, and scientific instrumentation are some other embedded applications that heavily rely on signal processing.

Applications for signal processing all have similar traits. They deal with a lot of data at first. The information might include samples from a physical processor taken at different points in time (such as wireless radio signal samples), samples from space (such as pictures), or samples from both (such as video and radar). Second, they often use advanced mathematical processes to the data, such as feature extraction, machine learning, frequency analysis, system identification, and filtering. These procedures need a lot of math.

Digital signal processors, or DSPs for short, are processors created especially to serve computationally demanding signal processing applications. It is important to comprehend the structure of standard signal processing algorithms in order to have some idea of the structure of such processors and the consequences for the designer of embedded software. Finite impulse response (FIR) filtering is a standard signal processing approach that is used in one way or another in all of the aforementioned applications. Even in its most basic version, this algorithm has significant effects on hardware. An input signal x in its most basic form consists of an extremely lengthy series of numerical values, which is so long that for design purposes The CPU must manage the data transfer down the delay line in addition to the many arithmetic operations, as seen.

DSP processors may implement one tap of a FIR filter in one cycle by supporting delay lines and multiply-accumulate instructions, as illustrated in Example 8.6. In that cycle, two values are multiplied, the product is added to an accumulator, and two pointers are incremented or decremented using modulo arithmetic.

Graphics processors

A graphics processing unit (GPU) is a unique kind of processor created specifically to carry out the computations necessary for generating visuals. These processors were first used to produce text and images in the 1970s. They were also combined with other visual patterns and used to create shapes including rectangles, triangles, circles, and arcs. Shaders, digital video, and 3D graphics are all supported by modern GPUs. Today's leading GPU manufacturers are Intel, NVIDIA, and AMD.

GPUs work well with certain embedded applications, especially gaming. Moreover, GPUs have progressed towards more broad programming paradigms and are already showing up in other computationally demanding applications, including instrumentation. Due to their inherent high-power consumption, GPUs are currently not a suitable fit for embedded systems that must save energy.

Parallelism

Today's CPUs provide a variety of parallelism options. The time of a program's execution is greatly influenced by these factors, therefore embedded system designers must be aware of them. A summary of the various forms and their implications for system designers are given in this section. The foundation of embedded systems is concurrency. Several sections of a computer programme are said to be concurrent if they theoretically run concurrently. If several software components really run concurrently, the programme is said to be parallel. If the target machine supports it, a compiler may examine the relationships between actions in a programme and generate parallel code. Dataflow analysis is the name given to this study. Using VLIW (very long instruction word) architectures or multi-issue instruction streams, many microprocessors now offer parallel processing. Independent instructions may be carried out concurrently by processors having multi-issue instruction streams. Figure 1 illustrate the embedded CPU.

When there are no dependencies, the hardware executes many instructions at once after on-the-fly dependency analysis. In the latter, assemblylevel instructions on VLIW computers indicate numerous operations to be carried out simultaneously. The compiler is often necessary to generate the proper parallel instructions in this situation. In these situations, the dependency analysis is performed at the assembly language or individual operation level rather than the C line level. Several actions, or even more sophisticated ones like procedure calls, may be specified in a single line of C code. An imperative programme is examined for concurrency in both scenarios (multi-issue and VLIW) to allow for concurrent execution. Increasing programme execution speed is the main goal. The objective is enhanced performance, and it is assumed that completing a work sooner rather than later is always preferable.

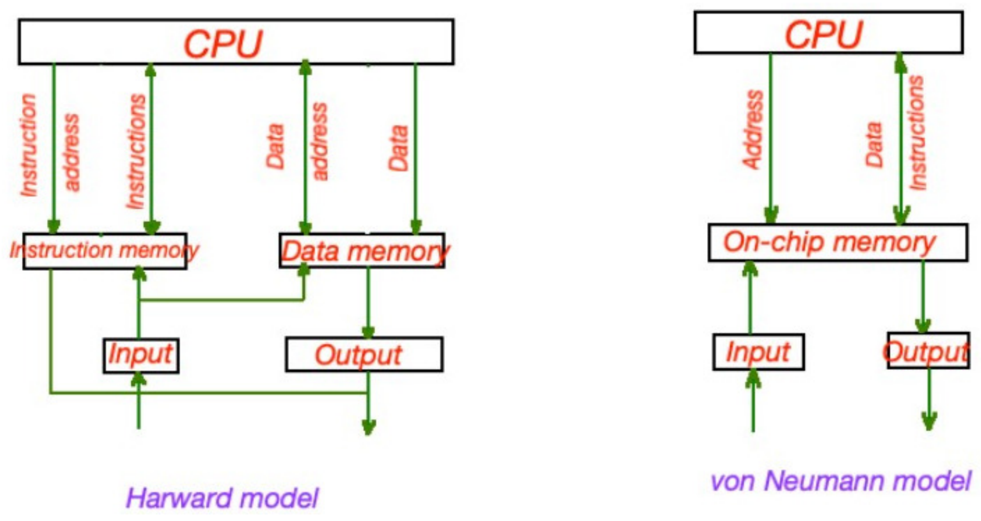


Figure 1: Illustratethe embedded CPU and memory.

Concurrency, however, has a role in embedded systems that is considerably more important than just enhancing speed. Physical processes and embedded programmes interact, and concurrent activity is a feature of the physical world. Embedded programmes often have to keep track of, respond to, and concurrently operate a number of output devices that have an impact on the real environment.

Concurrency is a fundamental component of the logic of embedded systems, which are nearly invariably concurrent programmes. It is more than simply a means of achieving better performance. In fact, completing a job sooner rather than later does not always make sense. Naturally, timing is important; acts taken in the physical world often need to be taken at the appropriate moment neither early nor late. Imagine, for instance, a gasoline engine controller. There is no doubt that firing the spark plugs early is not preferable to firing them later.

Concurrent programmes may also be run sequentially or concurrently, much as imperative programmes. A multitasking operating system normally performs the sequential execution of a concurrent programme today, interleaving the execution of many tasks in a single sequential stream of instructions. Of course, if the CPU has a multi-issue or VLIW architecture, the hardware may parallelize that execution. As a result, an operating system may turn a concurrent programme into a sequential stream, and the hardware may then convert it back to a concurrent programme in order to increase performance. The challenge of making sure that things happen at the proper moment is substantially complicated by these many translations. Chapter 12 addresses this concern.

The major topic of this chapter is parallelism in hardware, which exists to enhance performance for applications that need lots of computing. From the standpoint of the programmer, concurrency results from the hardware created to enhance performance, not from the resolution of the application issue. In other words, the application just requires that things be done rapidly, not (necessarily) that many operations take place at once. Hence, many intriguing applications that result from parallelism and application needs will incorporate both types of concurrency.The types of algorithms used in compute-intensive embedded applications have a significant impact on the hardware's architecture. In this part, pipelining, instruction-level parallelism, and multicore architectures are the hardware strategies we concentrate on to achieve parallelism. All of these have a significant impact on the embedded software programming paradigms.

CONCLUSION

Embedded processors are a crucial technology that has enabled the development of many innovative applications and devices, from smartphones and smart watches to medical devices and industrial equipment. Embedded processors come in a range of sizes, power consumption, and performance, and have to balance the trade-off between these factors for optimal performance, power consumption, and cost. The ongoing advancements in embedded processors, including AI and machine learning, security and privacy, energy efficiency, and specialized applications, are likely to shape the future of the technology. These developments will enable the creation of new applications and technologies that have the potential to transform various sectors and change the way we live and work.

REFERENCES:

[1] K. Chen, Q. Deng, Y. Hou, Y. Jin, and X. Guo, "Hardware and software co-verification from security perspective," in *Proceedings - 2019 20th International Workshop on Microprocessor/SoC Test, Security and Verification, MTV 2019*, 2019. doi: 10.1109/MTV48867.2019.00018.

[2] B. S. Naik, Y. Suresh, J. Venkataramanaiah, and A. K. Panda, "Design and implementation of a novel nine-level MT-MLI with a self-voltage-balancing switching technique," *IET Power Electron.*, 2019, doi: 10.1049/iet-pel.2018.6119.

[3] V. Muttillo, P. Giammatteo, V. Stoico, and L. Pomante, "An early-stage statement-level metric for energy characterization of embedded processors," *Microprocess. Microsyst.*, 2020, doi: 10.1016/j.micpro.2020.103200.

[4] M. Yang and Y. Hara-Azumi, "Implementation of Lightweight eHealth Applications on a Low-Power Embedded Processor," *IEEE Access*, 2020, doi: 10.1109/ACCESS.2020.3006901.

[5] M. Ebrahimi *et al.*, "Comprehensive Analysis of Sequential and Combinational Soft Errors in an Embedded Processor," *IEEE Trans. Comput. Des. Integr. Circuits Syst.*, 2015, doi: 10.1109/TCAD.2015.2422845.

[6] X. Wang *et al.*, "Two-stage checkpoint based security monitoring and fault recovery architecture for embedded processor," *Electron.*, 2020, doi: 10.3390/electronics9071165.

[7] H. Seo, H. Kwon, H. Kim, and J. Park, "Ace: Aria-ctr encryption for low-end embedded processors," *Sensors (Switzerland)*, 2020, doi: 10.3390/s20133788.

[8] S. Kurniawan, D. K. Halim, H. Dicky, and C. M. Tang, "Multicore development environment for embedded processor in arduino IDE," *Telkomnika (Telecommunication Comput. Electron. Control.*, 2020, doi: 10.12928/TELKOMNIKA.V18I2.14873.

CHAPTER 11

THE MEMORY ARCHITECTURE ASSIGNMENT SCHEMES

Mr. Aishwary Awasthi, Research Scholar
Department of Mechanical Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- aishwary@sanskriti.edu.in

ABSTRACT:

Memory architectures play a crucial role in the performance and efficiency of computer systems. A memory architecture refers to the organization of the memory subsystem in a computer system, including the types of memory used, the organization of memory banks, and the access methods used by the system we will explore the various types of memory architectures, including single-channel, dual-channel, and multi-channel memory architectures. We will also discuss the benefits and drawbacks of each type of memory architecture, as well as their impact on system performance and efficiency.

KEYWORDS:

Computer System, Dual-Channel, Multi-Channel, Memory Architecture, Single-Channel.

INTRODUCTION

Memory is an essential component of modern computing systems, and it is used to store data and instructions that are used by the processor. There are several types of memory architectures, each with a unique set of features and applications. This article provides an overview of memory architectures and their use in modern computing systems.

Memory Hierarchy:

Memory hierarchy is a system of levels of memory that are arranged in a hierarchy based on speed, cost, and capacity. The memory hierarchy consists of four levels: registers, cache, main memory, and secondary memory.

1. **Registers:** Registers are the fastest and smallest type of memory in a computing system. They are used to store data and instructions that are currently being processed by the processor.
2. **Cache:** Cache is a small, fast memory that is used to store frequently accessed data and instructions. Cache memory is placed between the processor and main memory, and it is used to reduce the latency of accessing data from main memory.
3. **Main Memory:** Main memory is the primary memory of a computing system. It is used to store data and instructions that are not currently being processed by the processor. Main memory is larger and slower than cache memory, but it has a larger capacity.
4. **Secondary Memory:** Secondary memory is a type of memory that is used to store data and instructions that are not currently being used by the processor. Secondary memory is typically slower and has a larger capacity than main memory. Examples of secondary memory include hard disk drives, solid-state drives, and magnetic tape.

Memory Architectures:

There are several types of memory architectures, each with a unique set of features and applications. The following are some of the most common types of memory architectures:

1. **Von Neumann Architecture:** The Von Neumann architecture is a type of memory architecture that is used in most modern computing systems [1]. It is named after John von Neumann, who first proposed the architecture in the 1940s. The Von Neumann architecture uses a single bus to transfer data and instructions between the processor and memory. It is a sequential architecture, which means that it processes data and instructions one at a time.
2. **Harvard Architecture:** The Harvard architecture is a type of memory architecture that uses separate buses for instructions and data. It is named after Harvard University, where it was first developed in the 1930s. The Harvard architecture is a parallel architecture, which means that it can process multiple instructions simultaneously. The Harvard architecture is commonly used in embedded systems and digital signal processing applications.
3. **Modified Harvard Architecture:** The modified Harvard architecture is a type of memory architecture that is a hybrid of the Von Neumann and Harvard architectures. It uses separate buses for instructions and data, like the Harvard architecture, but it also allows data to be stored in the same memory space as instructions, like the Von Neumann architecture. The modified Harvard architecture is commonly used in digital signal processing and multimedia applications.
4. **Associative Memory:** Associative memory is a type of memory architecture that allows data to be accessed based on its content, rather than its location. Associative memory is commonly used in applications such as image and speech recognition, where data is searched based on its content.

Memory is an essential component of modern computing systems, and there are several types of memory architectures, each with a unique set of features and applications. Memory hierarchy is a system of levels of memory that are arranged in a hierarchy based on speed, cost, and capacity [2]. The memory hierarchy consists of registers, cache, main memory, and secondary memory. The choice of memory architecture depends on the specific requirements of the application, such as speed, cost, and capacity. The continued development of memory architectures will be critical for the advancement of various industries and the continued development of modern technology.

Types of Memory:

In addition to memory architectures, there are several types of memory, each with unique features and applications. The following are some of the most common types of memory:

1. **Random Access Memory (RAM):** RAM is a type of volatile memory that is used to store data and instructions that are currently being used by the processor. RAM is typically used as main memory in a computing system, and it is faster than secondary memory but slower than cache memory. RAM is divided into two types: dynamic RAM (DRAM) and static RAM (SRAM). DRAM is used for main memory in a computing system, while SRAM is used for cache memory.

2. **Read-Only Memory (ROM):** ROM is a type of non-volatile memory that is used to store instructions that are needed for booting up a computer or other computing system [3]. ROM is typically used for firmware, which is the low-level software that is stored in a computing system's memory.

3. **Flash Memory:** Flash memory is a type of non-volatile memory that is used in many types of computing devices, including digital cameras, MP3 players, and USB flash drives. Flash memory is similar to EEPROM (electrically erasable programmable read-only memory), but it can be erased and reprogrammed in blocks, rather than byte-by-byte.

4. **Magnetic Storage:** Magnetic storage is a type of secondary memory that is used in hard disk drives and magnetic tape. Magnetic storage uses a magnetic field to store data, and it is slower than solid-state storage but has a larger capacity.

5. **Solid-State Storage:** Solid-state storage is a type of secondary memory that is used in solid-state drives (SSDs) and memory cards. Solid-state storage uses semiconductor memory to store data, and it is faster than magnetic storage but has a smaller capacity.

Memory architectures and types of memory are critical components of modern computing systems. Memory hierarchy is a system of levels of memory that are arranged in a hierarchy based on speed, cost, and capacity. There are several types of memory architectures, each with unique features and applications, including Von Neumann, Harvard, Modified Harvard, and Associative Memory. Additionally, there are several types of memory, including RAM, ROM, Flash Memory, Magnetic Storage, and Solid-State Storage. The choice of memory architecture and type of memory depends on the specific requirements of the application, such as speed, cost, and capacity. As technology continues to advance, the development of new memory architectures and types of memory will be critical for the continued advancement of modern computing systems.

Recent advancements in packaging technology allow DRAM chips to be stacked within processor packages or on top of processor chips, which may reduce the cost of DRAM per bit, provide broader interfaces, and increase bandwidth. To balance the tradeoffs between cost, performance, and capacity, system designers must take into account the fact that these technologies have lower performance and higher prices than conventional off-package memory. The most straightforward way to achieve this balance is to use a heterogeneous memory architecture that uses both on- and off-package memory. Designers must then choose whether to use the on-package memory as a NUMA memory peer to the off-package DRAM or as an extra cache-hierarchy level (managed by hardware or software). In order to explore memory hierarchies that include diverse memory technologies with various properties, this article gives a model and analysis of energy, bandwidth, and latency for existing and upcoming DRAM technologies. According to the research, there is a smaller difference between on- and off-package DRAM technologies than there is between the cache tiers of conventional memory architectures. Hence, in order to maintain system energy and bandwidth efficiency, heterogeneous memory caches must achieve extremely high hit rates [4].

Computer memories still fulfil the purpose for which they were first created in the electronic discrete variable automatic computer (EDVAC) device described by John von Neumann, namely the timely provision of instructions and operands for computations. The relative distance in processor cycles of this memory has dramatically risen as technology has made it feasible for computers with many processors to be much quicker and bigger. Microarchitectural techniques have advanced to share this memory across ever-larger systems of processors with deep cache hierarchies, and have hidden this latency for many applications. However, they are proving to be costly and energy-inefficient for newer types of problems working on massive amounts of data. In-memory databases that hold data in memory for much longer than the period of a single activity, scale-out systems dispersed over hundreds or even thousands of nodes, and near-data computing where part of the processing is off-loaded to the location of the data are examples of new paradigms. This article offers a historical perspective on the development of memory architecture and makes the case that the needs of novel issues and novel applications will probably radically alter the design of processors and systems in contrast to the well-established von Neumann model [5].

Modern electronic systems now face greater demands for quicker applications and lower power consumption due to rapidly advancing technology, faster and more powerful internet connections, and the growing use of mobile devices. RAM is just as efficient as a CPU in terms of performance and power consumption in contemporary electronic devices. While DRAM is now the most popular form of primary memory, it has not been able to keep up with the demands that have been growing. Improving the performance and power consumption of DRAM is one of the concerns that has to be addressed. The creation of hybrid main memory architectures is another investigation to answer this rising requirement. One of the most current RAM experiments is Hybrid Main Memory. For a more effective main memory design, we analyse hybrid main memory systems in this study [6].

The potential for concurrently optimising network accuracy and hardware efficiency makes the co-exploration of neural architectures and hardware design exciting. Yet for the standard von-Neumann computer architecture, whose speed is severely constrained by the well-known memory wall, cutting-edge neural architecture search algorithms are created specifically for the co-exploration. We are the first to combine the neural architecture search with computing-in-memory architecture in this research with the goal of identifying the most effective neural architectures with high network accuracy and optimal hardware efficiency. A lot of potential to improve performance come with such a new mix, but there are also many difficulties: The performance of the neural network may be severely harmed by device variation since the optimization space encompasses many design levels, from device type and circuit topology to neural architecture. To overcome these difficulties, we put forth the cross-layer exploration framework NACIM, which explores the design space for devices, circuits, and architectures while taking device variation into account in order to identify the most durable neural architectures paired with the most effective hardware design[7].

Minimizing the energy consumption of the L2 cache, main memory, and interconnects to that memory is one of the most difficult challenges in exascale computing. This challenge is made more difficult by the cryogenic system requirements that reveal the absence of high-density, high-speed, and power-efficient memory in the technology for potential cryogenic computing systems using Josephson junction superconducting logic. Here, we show an array of cryogenic memory cells made up of a superconducting heater-cryotron bit-select element and a non-volatile three-terminal magnetic tunnel junction element that is powered by the spin Hall effect. These memory elements have a bit-select element and a write energy of around 8 pJ; they were created to have a minimal overhead power usage of about 30%. A 4 4 array may be completely addressed with bit select error rates of 106, while individual magnetic memory cells evaluated at 4 K exhibit reliable switching with write error rates of 106. This demonstration is the first step towards a complete cryogenic memory architecture that aims to meet energy and performance standards suitable for use in superconducting high performance and quantum computing control systems, both of which need large amounts of memory working at 4 K [8].

The need of using a lot of data to keep efficiency high. The performance of algorithms is greatly impacted by memory bandwidth constraints and the performance gap between memory and logic, which raises the total time and energy needed for processing. Logic-In-Memory is a potential method to get around these restrictions (LIM). In this study, we provide a non-volatile skyrmion-based recetrack memory-based LIM architecture. The architecture may be used as a memory or to carry out sophisticated logic operations on the data that is stored, such as finding the maximum/minimum number. Physical simulations for the memory array and digital design tools for the control logic were used in the circuit's design and validation. The results emphasise the suggested architecture's modest size and strong energy efficiency when compared to a reference CMOS implementation.

DISCUSSION

Memory architectures refer to the organization of the memory subsystem in a computer system, including the types of memory used, the organization of memory banks, and the access methods used by the system. The design of a memory architecture is a critical aspect of computer system design and can have a significant impact on the performance and efficiency of the system. Memory architecture has evolved over the years to meet the changing needs of computer systems. From the early days of computers when memory was limited and expensive, to today's computers with a wide range of memory technologies available, memory architecture has gone through significant changes. We will explore the various types of memory architectures used in computer systems, the benefits and drawbacks of each type, and their impact on system performance and efficiency. We will also discuss the ongoing developments and advancements in memory architecture and their potential impact on the future of computer systems.

Types of Memory Architectures

There are several types of memory architectures used in computer systems, including:

Single-Channel Memory Architecture

The single-channel memory architecture is the simplest and most basic memory architecture used in computers. It uses a single memory module that is accessed by the CPU through a single memory controller. This type of memory architecture is commonly used in low-end and entry-level systems and is generally less expensive than other memory architectures.

The primary advantage of a single-channel memory architecture is its simplicity, which makes it easy to implement and maintain. However, its performance is limited by the single data path between the CPU and memory, which can result in a bottleneck in systems with high memory usage.

Dual-Channel Memory Architecture

The dual-channel memory architecture is an enhancement over the single-channel memory architecture that uses two memory modules that are accessed by the CPU through two memory controllers. This type of memory architecture provides double the bandwidth of the single-channel architecture, which can improve system performance in applications that require high memory usage, such as gaming, video editing, and 3D modeling.

The dual-channel memory architecture is commonly used in mid-range and high-end systems and provides a good balance between performance and cost. However, it is more complex to implement than the single-channel architecture, and it requires more memory modules, which can increase the cost of the system.

Multi-Channel Memory Architecture

The multi-channel memory architecture is an extension of the dual-channel memory architecture that uses three or more memory modules that are accessed by the CPU through multiple memory controllers. This type of memory architecture provides even higher bandwidth than the dual-channel architecture and is commonly used in high-performance systems, such as servers, workstations, and high-end gaming systems.

The primary advantage of a multi-channel memory architecture is its high performance, which makes it ideal for applications that require high memory usage and low latency. However, it is more complex and expensive to implement than the single-channel and dual-channel architectures, which can limit its use in low-end and entry-level systems.

Symmetric Memory Architecture

The symmetric memory architecture is a type of memory architecture that provides equal access to memory for all processors in a system. In a symmetric memory architecture, all processors have access to a shared memory pool, which can be accessed through a single memory controller or multiple memory controllers.

The symmetric memory architecture is commonly used in multiprocessor systems, such as servers and supercomputers, and provides high scalability and flexibility. However, its performance is limited by the memory bandwidth, which can result in contention and performance bottlenecks in systems with high memory usage.

Asymmetric Memory Architecture

The asymmetric memory architecture is a type of memory architecture that provides different amounts of memory to different processors in a system. In an asymmetric memory architecture, each processor has its memory pool, which can be accessed through a single memory controller or multiple memory controllers.

The asymmetric memory architecture is commonly used in systems with different processing requirements, such as high-performance computing clusters, where some nodes require more memory than others. This type of memory architecture provides high flexibility and scalability, as it allows the system to be optimized for different processing requirements. However, its performance is limited by the memory bandwidth, which can result in contention and performance bottlenecks in systems with high memory usage.

Memory Hierarchy Architecture

The memory hierarchy architecture is a type of memory architecture that uses multiple levels of memory, each with different performance characteristics, to provide a balance between performance and cost. In a memory hierarchy architecture, the CPU accesses the fastest and most expensive memory first, and then moves to slower and less expensive memory as needed. The memory hierarchy architecture is commonly used in modern computer systems and provides a good balance between performance and cost. It uses multiple levels of memory, including cache memory, main memory, and secondary storage, to provide a balance between the speed and cost of memory.

Benefits and Drawbacks of Different Memory Architectures

Different memory architectures offer different benefits and drawbacks, depending on the needs of the system. Some of the primary benefits and drawbacks of different memory architectures are:

Single-Channel Memory Architecture

Benefits:

a) Simplicity and ease of implementation

b) Lower cost than other memory architectures

Drawbacks:

a) Limited bandwidth, which can result in a bottleneck in systems with high memory usage

Dual-Channel Memory Architecture

Benefits:

a) Higher bandwidth than the single-channel architecture, which can improve system performance in applications that require high memory usage

b) Good balance between performance and cost

Drawbacks:

a) More complex and expensive to implement than the single-channel architecture

b) Requires more memory modules, which can increase the cost of the system

Multi-Channel Memory Architecture

Benefits:

a) High performance, which makes it ideal for applications that require high memory usage and low latency

b) Scalability and flexibility

Drawbacks:

a) More complex and expensive to implement than the single-channel and dual-channel architectures

b) Limited use in low-end and entry-level systems due to its cost

Symmetric Memory Architecture

Benefits:

a) Equal access to memory for all processors in a system

b) High scalability and flexibility

Drawbacks:

a) Performance limited by the memory bandwidth, which can result in contention and performance bottlenecks in systems with high memory usage

Asymmetric Memory Architecture

Benefits:

a) Different amounts of memory can be allocated to different processors, which allows the system to be optimized for different processing requirements

b) High flexibility and scalability

Drawbacks:

a) Performance limited by the memory bandwidth, which can result in contention and performance bottlenecks in systems with high memory usage

Memory Hierarchy Architecture

Benefits:

a) Good balance between performance and cost

b) Uses multiple levels of memory to provide a balance between the speed and cost of memory

Drawbacks:

a) Increased complexity due to the multiple levels of memory

b) Requires more hardware, which can increase the cost of the system

Advancements in Memory Architecture

Advancements in memory architecture are ongoing, with new technologies being developed to meet the changing needs of computer systems. Some of the recent advancements in memory architecture include:

Non-Volatile Memory

Non-volatile memory is a type of memory that retains its data even when the power is turned off. This type of memory is commonly used in secondary storage devices, such as hard disk drives and solid-state drives. However, recent developments in non-volatile memory technologies, such as phase-change memory and resistive random-access memory, have shown promise in replacing traditional volatile memory technologies, such as dynamic random-access memory.

Non-volatile memory technologies offer several advantages over volatile memory technologies, including lower power consumption, faster access times, and higher endurance. These advancements in non-volatile memory technologies could lead to significant improvements in the performance and efficiency of computer systems.

High-Bandwidth Memory

High-bandwidth memory (HBM) is a type of memory architecture that uses a stacked design to provide high memory bandwidth with a low power footprint. HBM is commonly used in graphics processing units (GPUs) and high-performance computing systems to provide high bandwidth for memory-intensive applications.

HBM uses a vertical stacking approach, where multiple layers of memory chips are stacked on top of each other and connected through vertical interconnects. This stacking approach allows for a significant increase in memory bandwidth, as the memory chips can be accessed simultaneously.

3D Memory

3D memory is a type of memory architecture that uses a stacking approach to increase the memory density and reduce the footprint of memory modules. 3D memory can be implemented using various technologies, such as through-silicon vias (TSVs) or microbumps. 3D memory can offer several benefits, including increased memory density, reduced power consumption, and improved system performance. 3D memory can be used in various applications, including mobile devices, data centers, and high-performance computing systems.

Hybrid Memory Cube

The hybrid memory cube (HMC) is a type of memory architecture that uses a 3D stacking approach to provide high bandwidth and low latency. HMC uses a vertical stacking approach, where multiple layers of memory chips are stacked on top of each other and connected through vertical interconnects. HMC provides high memory bandwidth with low power consumption, making it ideal for memory-intensive applications. HMC can be used in various applications, including high-performance computing systems, graphics processing units (GPUs), and networking equipment.

Several chip architects believe that memory systems, as opposed to data pipelines, have a greater influence on system performance overall. Of course, it depends on the application, but it holds true for a lot of them. Memory complexity mostly comes from three different places. Secondly, mixing several memory technologies into one embedded system is often essential. Several memory systems are volatile, which means that the

If electricity is removed, memory is also lost. The majority of embedded systems need both volatile and non-volatile memory. Moreover, there are a number of options available within these categories, and the choices have a big impact on the system designer. Second, since memories with higher capacities and/or lower power requirements are slower, memory hierarchy is often required. Faster memories must be combined with slower memories in order to attain appropriate performance at affordable cost. Thirdly, a CPU architecture's address space is segmented to provide access to different types of memory, to enable popular programming styles, and to assign addresses for interacting with devices other than memories, such I/O devices. We cover each of these three topics in turn in this chapter.

Memory problems are a major concern in embedded systems. The system designer must consider crucial repercussions while selecting memory technology. A programmer would need to consider, for instance, whether data will survive a power outage or the entry into a power-saving standby mode. Volatile memory is a kind of memory whose contents are lost when the power is turned off. We go through some of the possible technologies and their trade-offs in this section.

A microcomputer generally has some RAM (random access memory), which is a memory that allows individual things (bytes or words) to be written and retrieved one at a time reasonably fast. This memory is in addition to the register file. While DRAM (dynamic Memory) is quicker, SRAM (static RAM) is also bigger (each bit takes up more silicon area).

SRAM retains data as long as power is present, unlike DRAM, which can only store data for a limited length of time before needing to be refreshed. Both kinds of memories are volatile memory since they both lose their contents if power is absent, however some could argue that DRAM is more volatile than SRAM due to the fact that it does so even while power is present.

The majority of embedded computer systems include SRAM memory. Since it might be impossible to supply adequate memory using SRAM technology alone, many additionally integrate DRAM. While accessing memory, a programmer must be aware of whether the address is mapped to SRAM or DRAM if programme execution speed is a problem. More240 Introduction to Embedded Systems, Lee & Seshia

Moreover, since the Memory may be occupied with a refresh at the time the access is required, the refresh cycle of DRAM may bring unpredictability to the access timings. Access times might also be impacted by past access. The last memory address that was visited may affect how long it takes to access one memory location.

Each memory location must be refreshed, say, every 64 ms, and a group of locations (a "row") must be refreshed at the same time, according to the DRAM memory chip manufacturer's specifications. DRAM must be used with a controller that makes sure that all locations are refreshed frequently enough to keep the data from being lost. The mere act of reading the memory will refresh the locations that are read (and locations on the same row), but since applications might not access all rows within the allotted time period. If the memory is occupied with a refresh when the access is started, the memory controller will delay the access. This throws uncertainty into the program's timeline.

Data storage is a need for embedded devices even when the power is off. For this, there are several possibilities. Of course, one is to provide a battery backup so that there is never a power outage. Batteries, however, eventually run out of power, and non-volatile memories offer a superior alternative. Magnetic core memory, often known as core, was a sort of early non-volatile memory in which a ferromagnetic ring was magnetised to store data. Even if multicore processors are becoming more common, the word "core" is still used in computing to describe computer memory.

Nowadays, ROM (read-only memory) or mask ROM, whose contents are set at the chip manufacturing, is the most fundamental kind of non-volatile memory. This may be helpful for mass-produced devices if all that has to be saved is a software and continuous data that never changes. Firmware is the term used for these programmes, implying that they are not as "soft" as software. The ability to programme ROM in the field comes in a variety of forms, and technology has advanced to the point that they are now almost always preferred over mask ROM. There are several versions of EEPROM, electrically-erasable programmable ROM, but they can all be written to. The write time normally takes significantly longer than the read time, and the device's lifespan is capped at a certain amount of writes. Flash memory is a kind of EEPROM that is very practical. Firmware and user data that must survive a power outage are often stored on flash.

While it is a particularly practical kind of non-volatile memory and was developed by Dr. Fujio Masuoka at Toshiba in the 1980s, flash memory poses some intriguing problems for Lee & Seshia, Introduction to Embedded Systems 241 9.2. designers of embedded systems using MEMORY HIERARCHY. Frequently accessed data will often need to be transferred from the flash to RAM before being utilised by a programme since flash memory typically have read times that are quite quick but not as quick as SRAM and DRAM. These memories are not a replacement for working memory since the write times are substantially longer than the read timings, and the total number of writes is limited.

Flash memory comes in two varieties: NOR flash and NAND flash. While NOR memory may be accessed similarly to RAM, it has longer erase and write times. Even though NAND flash is less costly and has quicker erase and write speeds, data must be read in blocks of hundreds to thousands of bits at a time. This indicates that it functions more like a secondary storage device, such as a hard drive or optical media, such as CD or DVD, from a system viewpoint. As of this writing, NOR memory and NAND flash can both only be wiped and overwritten a certain number of times typically, around 1,000,000 for NOR flash and under 10,000,000 for NAND storage.

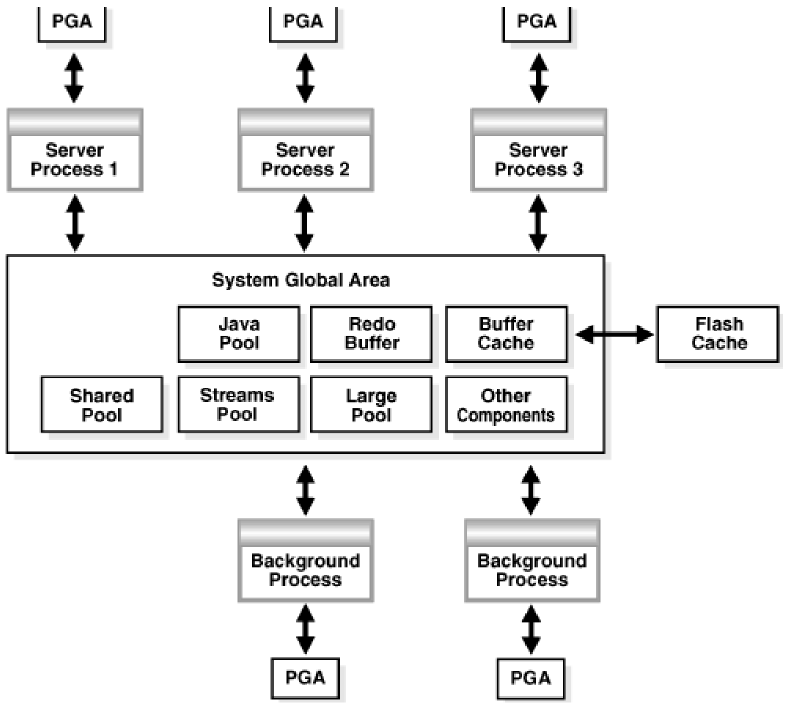


Figure 1: Illustrate the memory architecture.

For embedded system designers, the lengthier access times, constrained write counts, and block-wise accesses (for NAND memory) all add to the difficulty. Both hardware and software design must take into consideration these characteristics. Also, disc memories are non-volatile. While they may store enormous quantities of data, access times can grow significantly. In instance, the controller must wait until the head is over the desired place before the data at that point can be read due to the physics of a spinning disc and a read-write head. This process might take any amount of time. Disks are more challenging to employ in many embedded applications because they are more vibration-sensitive than the solid-state memory mentioned above. In Figure 1 shows the memory architecture.

More memory is needed for many applications than what is built into a microcomputer's on-chip memory. In order to enhance total memory capacity while minimising cost, latency, and energy usage, many processors use a memory hierarchy. In most cases, a smaller quantity of on-chip SRAM will be combined with more off-chip DRAM. They may be supplemented further with a third level, such as disc drives, which have a huge capacity but lack random access, making them relatively sluggish to read and write. It's possible that the application programmer is unaware of the memory fragmentation present in these technologies. The varied technology is made possible by a widely utilised technique called virtual memory.

To a programmer, memory architectures seem a continuous address space. Address translation, which transforms logical addresses in the address space into physical locations in one of the possible memory technologies, is provided by the operating system and/or the hardware. A specialised piece of hardware known as a translation lookaside buffer (TLB), which may accelerate certain address translations, often helps with this translation. These methods may present major issues for embedded system designers since it is exceedingly difficult to forecast or comprehend how long memory accesses would take. Hence, embedded system designers often need a deeper understanding of the memory system than general-purpose programmers.

Memory Maps a processor's memory map specifies how addresses are translated into hardware. The address width of the processor places a limit on the entire size of the address space. If each address corresponds to one byte, a 32-bit CPU, for instance, may address 2 32 locations, or 4 gigabytes (GB). With the exception of 8-bit processors, where the address width is often greater, the address width normally corresponds to the word width often 16 bits. For instance, the memory map of an ARM CortexTM-M3. Different structures will have different designs, but the general pattern will remain. Keep in mind that this design divides addresses used for data memory from those used for programme memory labelled A in the diagram B and D. This typical architecture enables various memories to be accessed through different buses, enabling the simultaneous retrieval of both instructions and data. The memory bandwidth is essentially doubled as a result. A Harvard architecture is one that separates programme memory from data memory in this way. Unlike the traditional von Neumann architecture, which keeps programmes and data in the same memory, it does not.

This memory map imposes restrictions on every given silicon implementation of this architecture. For instance, the ARM CortexTM - M3 core-equipped Luminary Micro1 LM3S8962 controller only has 256 KB of on-chip flash memory, much less than the architecture's maximum of 0.5 GB. The addresses 0x00000000 through 0x0003FFFF are used to represent this memory. The remaining addresses, 0x00040000 through 0x1FFFFFFF that the architecture permits for programme memory are "reserved addresses," which means that a compiler for this specific device shouldn't utilise them. The register file is where a processor's memory is most closely integrated. A word is kept in each register of the file. One essential aspect of a CPU architecture is word size. On an 8-bit architecture, it is one byte, on a 32-bit architecture, four bytes, and on a 64-bit architecture, eight bytes. The register file may be implemented directly in the CPU circuitry using flip flops, or the registers can be gathered into a single memory bank, often utilising the same SRAM technology as previously stated.

In most cases, a CPU has a modest number of registers. This is due more to the price of the bits in an instruction word than it is to the hardware cost of the register file. Instructions that may access one, two, or three registers are often provided via an instruction set architecture (ISA). These instructions must not use too many bits to encode them in order to store programmes in memory effectively, and they must not use too many bits to identify the registers either. Each reference to a register needs 4 bits if the register file has 16 registers. A total of 12 bits are required if an instruction may refer to three registers. For example, if a word of instructions has 16 bits, only 4 bits are left for additional information in the instruction, such as the instruction's identification, which must also be encoded in the instruction. This lets us know, for instance, whether the instruction calls for the addition or subtraction of two registers, with the result being placed in the third register.

Memory technologies are combined in many embedded applications. It is said that certain memories are "closer" to the processor than others because they are accessed earlier. For instance, working data is often temporarily stored in a nearby memory (SRAM) so that the application may utilise it. It is referred to as a scratchpad if the near memory has a unique set of addresses and the programme is in charge of transferring data into or out of it to the distant memory. It is referred to be a cache if the nearby memory copies data from the remote memory, with the hardware managing the copying to and from.

Cache memories provide some severe challenges since their timing behaviour might vary significantly in unpredictable ways for embedded systems with strict real-time restrictions. On the other hand, handling the data on a scratchpad memory automatically may be rather time-consuming for a programmer, and compiler-driven automated approaches are still in their infancy.

An architecture will generally enable a far wider address space than what can be stored in the processor's physical memory, as described in Section 9.2.1, with a virtual memory system being utilised to give the programmer the appearance of a continuous address space. Programs relate to logical addresses, which the memory management unit (MMU) converts to physical addresses if the CPU has one. For instance, a process may be permitted to access the logical addresses 0x60000000 to 0x9FFFFFFF (region D in the picture) for a total of 1 GB of accessible data memory using the memory map in Figure 9.1. Whatever amount of physical memory that is present in area B may be used by the MMU to implement a cache. The MMU checks to see whether the memory address provided by the application is cached in area B, and if it is, it translates the address and completes the fetch. If not, then we have a cache miss, and the MMU takes care of pulling data into the cache from the secondary memory (in region D) (area B). The MMU causes a page fault if the location is also absent in area D, which may lead to software managing the transport of data from the disc into the memory. The software is therefore given the impression of having a large amount of memory, but at the expense of memory access times being highly unpredictable. Depending on how the logical addresses are distributed across the physical memories, it is normal for memory access times to vary by a factor of 1000 or more.

CONCLUSION

Memory architecture plays a critical role in the performance and efficiency of computer systems. Different memory architectures offer different benefits and drawbacks, depending on the needs of the system. Single-channel, dual-channel, multi-channel, symmetric, and asymmetric memory architectures are commonly used in modern computer systems. The memory hierarchy architecture provides a good balance between performance and cost by using multiple levels of memory. Advancements in memory architecture, such as non-volatile memory, high-bandwidth memory, 3D memory, and hybrid memory cube, are ongoing and can lead to significant improvements in the performance and efficiency of computer systems. These advancements in memory architecture are critical to meeting the changing needs of computer systems, which require higher performance, lower power consumption, and increased memory density.

REFERENCES:

[1] M. Elvir, A. J. Gonzalez, C. Walls, and B. Wilder, "Remembering a Conversation - A Conversational Memory Architecture for Embodied Conversational Agents," *J. Intell. Syst.*, 2017, doi: 10.1515/jisys-2015-0094.

[2] S. Hong, W. O. Kwon, and M. H. Oh, "Hardware Implementation and Analysis of Gen-Z Protocol for Memory-Centric Architecture," *IEEE Access*, 2020, doi: 10.1109/ACCESS.2020.3008227.

[3] Y. Zhang *et al.*, "A survey of memory architecture for 3D chip multi-processors," *Microprocess. Microsyst.*, 2014, doi: 10.1016/j.micpro.2014.03.007.

[4] E. Bolotin, D. Nellans, O. Villa, M. O'Connor, A. Ramirez, and S. W. Keckler, "Designing Efficient Heterogeneous Memory Architectures," *IEEE Micro*, 2015, doi: 10.1109/MM.2015.72.

[5] R. Nair, "Evolution of Memory Architecture," *Proc. IEEE*, 2015, doi: 10.1109/JPROC.2015.2435018.

[6] Z. YILDIZ ÇAVDAR, İ. AVCI, M. KOCA, and A. SERTBAŞ, "A Survey of Hybrid Main Memory Architectures," *Sak. Univ. J. Sci.*, 2019, doi: 10.16984/saufenbilder.334645.

[7] S. Raikwar, L. Jijyabhau Wani, S. Arun Kumar, and M. Sreenivasulu Rao, "Hardware-in-the-Loop test automation of embedded systems for agricultural tractors," *Meas. J. Int. Meas. Confed.*, 2019, doi: 10.1016/j.measurement.2018.10.014.

[8] M. H. Nguyen *et al.*, "Cryogenic Memory Architecture Integrating Spin Hall Effect based Magnetic Memory and Superconductive Cryotron Devices," *Sci. Rep.*, 2020, doi: 10.1038/s41598-019-57137-9.

CHAPTER 12

A COMPARISON BETWEEN INPUT AND OUTPUT TYPE HARDWARE

Dr. Sundar Singh, Assistant Professor
Department of Computer Science Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- sundar@sanskriti.edu.in

ABSTRACT:

Input and output (I/O) hardware devices are essential components of modern computer systems that allow users to interact with the digital world. I/O hardware devices are used to input data, such as text and images, into the computer system, as well as output data in various forms, such as sound, text, and graphics.

The design and selection of I/O hardware devices are crucial in developing efficient and effective computer systems, and considerations such as device compatibility, reliability, and cost must be taken into account. With the continuing development of technology, the role and capabilities of I/O hardware devices will continue to evolve, allowing for greater user engagement and more immersive experiences.

KEYWORDS:

Input device, Output Device, Hardware, Software, Graphics.

INTRODUCTION

Input and output hardware devices are an essential part of a computer system. Input devices allow users to provide data or commands to the computer system, while output devices display or print data processed by the computer. In this, we will discuss various types of input and output hardware devices, their functions, and how they work.

Input Hardware Devices:

1. **Keyboard:**

The keyboard is the most commonly used input device that allows users to enter data or commands into a computer system. It has a set of keys that correspond to various characters, numbers, and symbols. Users can enter data by pressing these keys [1]. There are different types of keyboards available in the market, such as a standard keyboard, ergonomic keyboard, gaming keyboard, and virtual keyboard.

2. **Mouse:**

A mouse is a pointing device that allows users to move a cursor on the screen and select objects. It has two or three buttons and a scroll wheel that helps users navigate through the computer system. There are different types of mice available, such as a standard mouse, wireless mouse, trackball mouse, and touchpad.

3. **Scanner:**

A scanner is an input device that allows users to scan and digitize images, documents, and other physical media. It works by capturing the image or document and converting it into a digital format that can be stored on a computer system. There are different types of scanners available, such as flatbed scanners, sheet-fed scanners, and handheld scanners.

4. **Microphone:**

A microphone is an input device that allows users to record sound or voice. It works by converting sound waves into an electrical signal that can be stored or processed by a computer system. There are different types of microphones available, such as a dynamic microphone, condenser microphone, and USB microphone.

5. **Webcam:**

A webcam is an input device that allows users to capture and record video. It works by capturing the video and converting it into a digital format that can be stored or processed by a computer system [2], [3]. There are different types of webcams available, such as a standard webcam, 4K webcam, and wireless webcam.

Output Hardware Devices:

1. **Monitor:**

A monitor is the most commonly used output device that displays data processed by the computer system. It works by converting digital signals into visual images that can be displayed on the screen. There are different types of monitors available, such as a standard monitor, curved monitor, and gaming monitor.

2. **Printer:**

A printer is an output device that allows users to print data processed by the computer system. It works by converting digital data into a printable format that can be printed on paper or other media. There are different types of printers available, such as a laser printer, inkjet printer, and 3D printer.

3. **Speaker:**

A speaker is an output device that allows users to listen to sound or music processed by the computer system. It works by converting electrical signals into sound waves that can be heard by humans. There are different types of speakers available, such as a desktop speaker, soundbar, and Bluetooth speaker.

4. **Projector:**

A projector is an output device that allows users to display data processed by the computer system on a large screen or wall. It works by projecting light through an image or video and displaying it on a surface. There are different types of projectors available, such as a standard projector, HD projector, and 4K projector.

5. **Headphones:**

Headphones are an output device that allows users to listen to sound or music processed by the computer system without disturbing others [4], [5]. It works by converting electrical signals into sound waves that can be heard by humans through a set of speakers attached to the ears. There are different types of headphones available, such as a wired headphone, wireless headphone, and noise-canceling headphone.

How Input and Output Hardware Devices Work. Now, let's delve deeper into how input and output hardware devices work.

1. **Keyboard:**

A keyboard is a set of keys that correspond to various characters, numbers, and symbols. When a user presses a key, it sends a signal to the computer system that identifies the key pressed. The computer system then processes the signal and performs the corresponding action or enters the corresponding character or number.

2. **Mouse:**

A mouse is a pointing device that allows users to move a cursor on the screen and select objects. When a user moves the mouse, it sends a signal to the computer system that identifies the direction and speed of the movement. The computer system then moves the cursor on the screen according to the mouse movement. When a user clicks a button on the mouse, it sends a signal to the computer system that performs the corresponding action, such as selecting an object or opening a menu.

3. **Scanner:**

A scanner is an input device that captures and digitizes images, documents, and other physical media. When a user places the media on the scanner bed and presses the scan button, the scanner captures the image or document and converts it into a digital format, such as a JPEG or PDF file. The digital file can then be saved on the computer system or printed.

4. **Microphone:**

A microphone is an input device that converts sound waves into an electrical signal that can be stored or processed by a computer system. When a user speaks or makes a sound near the microphone, it captures the sound waves and converts them into an electrical signal. The computer system then processes the electrical signal and stores or uses it in various applications, such as recording or voice recognition.

5. **Webcam:**

A webcam is an input device that captures and records video. When a user starts the webcam, it captures the video and converts it into a digital format, such as a MP4 or AVI file. The digital file can then be saved on the computer system or shared with others through various applications, such as video conferencing or social media.

6. **Monitor:**

A monitor is an output device that displays data processed by the computer system. When the computer system sends digital signals to the monitor, it converts the signals into visual images that can be displayed on the screen[6], [7]. The resolution and color of the image depend on the quality of the monitor and the graphics processing unit of the computer system.

7. **Printer:**

A printer is an output device that prints data processed by the computer system. When the computer system sends digital data to the printer, it converts the data into a printable format and prints it on paper or other media. The quality of the print depends on the quality of the printer, the ink or toner used, and the type of paper or media used.

8. **Speaker:**

A speaker is an output device that converts electrical signals into sound waves that can be heard by humans. When the computer system sends electrical signals to the speaker, it converts the signals into sound waves that can be heard through a set of speakers. The quality and volume of the sound depend on the quality of the speaker and the sound processing unit of the computer system.

9. **Projector:**

A projector is an output device that displays data processed by the computer system on a large screen or wall. When the computer system sends digital signals to the projector, it projects light through an image or video and displays it on a surface. The quality of the projection depends on the quality of the projector, the resolution of the image or video, and the lighting conditions of the environment.

10. **Headphones:**

Headphones are an output device that allows users to listen to sound or music processed by the computer system without disturbing others. When the computer system sends electrical signals to the headphones, it converts the signals into sound waves that can be heard through a set of speakers attached to

Input and output hardware devices are essential components of modern computer systems. They enable users to interact with a computer system by providing input commands and receiving output information. The input devices are used to input data into a computer system, while output devices are used to receive output information from the system. This article will discuss various input and output hardware devices in detail.

Input Devices:

Input devices are hardware components that allow the user to input data into a computer system. These devices translate human inputs into machine-readable language. Input devices come in many forms, ranging from the traditional keyboard and mouse to the more advanced touchscreens, voice recognition systems, and biometric scanners.

1. **Keyboard:** A keyboard is a standard input device that uses a set of keys to input data into a computer system. The keys are arranged in a specific layout, such as the QWERTY

layout, and can be used to type letters, numbers, symbols, and commands. Most keyboards also include additional keys for special functions, such as multimedia control, volume control, and sleep mode.

2. **Mouse:** A mouse is a small hand-held device that is used to move a cursor on the computer screen and to select, drag, and drop objects. A mouse typically has one or more buttons that can be clicked to initiate an action. Modern mice often include additional features such as a scroll wheel, back and forward buttons, and touchpads.
3. **Touchscreen:** A touchscreen is an input device that allows the user to interact directly with the computer screen. Touchscreens are commonly used in mobile devices such as smartphones and tablets, as well as in kiosks and interactive displays. The user can interact with the touchscreen by tapping, swiping, and pinching the screen.
4. **Digital Pen:** A digital pen, also known as a stylus, is an input device that is used to write, draw, or select on a digital surface. Digital pens can be used with a variety of devices, including tablets, smartphones, and touchscreens. They often include pressure-sensitive technology that allows for precise control over the digital ink.
5. **Scanner:** A scanner is an input device that is used to create a digital image of a physical document or object. Scanners can be flatbed, sheet-fed, or handheld and can be used to scan photos, documents, and objects such as barcodes and QR codes.
6. **Webcam:** A webcam is a small camera that is used to capture video and images. Webcams are commonly used in video conferencing, online streaming, and for taking selfies.
7. **Microphone:** A microphone is an input device that is used to capture audio. Microphones can be built into a computer system or can be external devices. They are commonly used in voice recognition systems, video conferencing, and for recording audio.
8. **Biometric Scanner:** A biometric scanner is an input device that is used to identify a person based on unique physical characteristics, such as fingerprints, iris patterns, or facial features. Biometric scanners are commonly used in security systems, access control systems, and for identity verification.

Output Devices:

Output devices are hardware components that allow the computer system to communicate with the user by providing output information [8]. These devices translate machine language into human-readable language. Output devices come in many forms, ranging from the traditional monitor and printer to more advanced devices such as projectors and speakers.

1. **Monitor:** A monitor is an output device that displays visual information on a computer screen. Monitors come in various sizes, resolutions, and technologies, including LCD, LED, and OLED. They can be used for various applications, including gaming, graphic design, and video editing.
2. **Printer:** A printer is an output device that prints text and images on paper. Printers can be inkjet, laser, or dot-matrix and can be used to print a variety of materials, including

documents, photos, and labels. Some printers also have additional functions, such as scanning, copying, and faxing.

3. **Projector:** A projector is an output device that projects visual information onto a larger screen or surface. Projectors are commonly used for presentations, movie screenings, and in classrooms and lecture halls. They come in various types, including LCD, DLP, and LCoS, and can be used with various devices, such as laptops, DVD players, and gaming consoles.
4. **Speakers:** Speakers are output devices that produce sound. They can be built into a computer system or can be external devices. Speakers are commonly used for listening to music, watching videos, and for gaming.
5. **Headphones:** Headphones are output devices that are worn over the ears to listen to audio without disturbing others. They can be wired or wireless and come in various styles and designs. Headphones are commonly used for listening to music, watching videos, and for gaming.
6. **Sound Card:** A sound card is an internal or external device that allows a computer system to produce sound. It can be used to improve the quality of audio playback, record audio, and to provide additional audio features such as surround sound.
7. **Plotter:** A plotter is an output device that is used to draw vector graphics. It is commonly used in engineering, architecture, and graphic design to create precise, large-scale drawings. Plotters come in various types, including drum plotters, flatbed plotters, and pen plotters.
8. **Braille Display:** A Braille display is an output device that is used to display Braille characters for blind or visually impaired users. It converts text from a computer system into Braille characters, allowing the user to read the text through touch.

Input and output hardware devices play an essential role in modern computer systems. They enable users to interact with the system, input data, and receive output information. The various input and output devices discussed in this article cater to different needs and preferences, and the selection of the right devices depends on the user's requirements and the intended use of the computer system.

Since computational and physical dynamics are combined in cyber-physical systems, any design must prioritise the mechanisms in processors that enable interaction with the outside environment. A system designer must deal with a variety of problems. The interfaces' mechanical and electrical characteristics are significant among them. Inappropriate usage of components, such as pulling excessive current from a pin, may result in system failure or other problems minimize the time it is helpful. Also, a lot of things occur simultaneously in the physical world.

Software, on the other hand, is often sequential. In the design of embedded systems, balancing these two incompatible features is often the main risk factor and a significant difficulty. Dramatic system failures may result from improper interactions between sequential programming and concurrent physical occurrences. In this chapter, we discuss problems.

Embedded processors, whether they are general-purpose processors, DSP processors, or microcontrollers, often include a variety of input and output (I/O) methods available to designers as pins on the chip. In this part, we go through some of the most popular interfaces available and use the following running example to illustrate their characteristics.

Under the graphical display in the middle lies the actual microcontroller. Several of the microcontroller's pins are accessible at the connections that are visible at the board's top and bottom as well as on each side of the microcontroller. A bespoke circuit board that only contains the hardware needed by the application would normally replace such a board in the finished product after being used to prototype an embedded application. With an integrated development environment (IDE) given by the manufacturer, an engineer will create the software for the board and put it onto flash memory that will be placed into the board's bottom slot. As an alternative, software may be uploaded to the board from the development computer via the USB port at the top.

In the previous illustration, the evaluation board is more than just a CPU since it also has a display and other hardware connectors switches and a speaker, are example. A single-board computer or microcomputer board are common names for such a board. The interfaces that a microcontroller or single-board computer offers are then briefly discussed.

A method for effectively sending a changeable quantity of power to external hardware devices is pulse width modulation (PWM). For instance, it may be used to regulate the temperature of a heating element, the brightness of an LED light, and the speed of electric motors. It can generally give a range of power to equipment that can handle quick and abrupt variations in voltage and current. PWM technology is simple to incorporate on the same chip as a microcontroller since it just requires digital circuitry. By design, digital circuits can generate only the high and low voltage levels. A PWM signal quickly alternates between high and low at a set frequency while altering the length of time the signal is high. The ratio is the duty cycle.

PWM peripheral devices are offered by several microcontrollers. A programmer often enters a value into a memory-mapped register to utilise them by setting the duty cycle the frequency may also be settable. After then, the gadget provides power in accordance to the desired duty cycle to external devices. When more current flows through a resistor, for instance, its temperature rises, acting as a heating element. Since temperature fluctuates more slowly than a PWM signal does, the resistor averages out the signal's quickly changing voltage, keeping the temperature extremely near to constant for a set duty cycle. Motors also smooth out input voltage changes that occur quickly. Likewise, LED and incandescent lights. Any device that responds slowly to changes in current or voltage relative to the PWM signal's frequency is a candidate for PWM control.

Specialized or customised digital hardware must commonly be connected to embed processors by embedded system designers. Several general-purpose I/O pins (GPIO) are included on many embedded CPUs, allowing software to read or write voltage levels denoting a logical zero or one. In active high logic, if the processor supply voltage is VDD, a value near VDD represents a logical one, whereas a voltage near zero represents a logical zero. Both of these interpretations are incorrect in active low logic. A GPIO pin may be set up as an output in various configurations. This allows software to adjust the output voltage to either high or low by writing to a memory-mapped register. Software may directly control external physical devices via this approach.

So, use cautious. An interface designer must be familiar with the device's specs when integrating devices using GPIO pins. Each gadget has a different voltage and current level. When attaching a device to a GPIO pin that, when given a logical one, provides an output voltage of VDD, the designer must be aware of the current restrictions. Ohm's rule states that the output current will be I = VDD/R if, for example, a device with a resistance of R ohms is connected to it.

This must remain updated while staying within predetermined limitations. If these limits are exceeded, the gadget can overheat and stop working. To produce sufficient current, a power amplifier could be required. Altering voltage levels may also need the use of an amplifier. The GPIO pins of the Luminary Micro Stellaris R microcontroller may be set up to source or sink a range of currents up to 18 mA. There are limitations on which pin combinations may support such comparatively large currents. Luminary Micro R (2008b), for instance, specifies that "the high-current GPIO package pins must be chosen so that there are only a maximum of two per side of the physical package... with the overall number of high-current GPIO outputs not exceeding four for the whole package." These limitations are created to stop the gadget from overheating.

Maintaining electrical separation between CPU circuits and external devices may also be crucial. The processor may become unstable if the electrical properties of the external devices are disorderly (noisy) and leak into the processor's power or ground connections. Alternately, the external device could run under extremely different voltage or power conditions than the CPU. Creating electrical domains inside a circuit that are generally independent of one another, maybe with distinct power supply, is a helpful technique. Transformers and opto-isolators are examples of isolation devices that may be utilised to permit communication between electrical domains. In the former, an electrical signal is converted into light in one electrical domain, and light is detected in another electrical domain and converted back into an electrical signal there. The latter use electrical domain-to-domain inductive connection.

Moreover, GPIO pins may be set up as inputs, enabling software to respond to externally supplied voltage levels. Schmitt triggers may cause input pins to exhibit hysteresis, much as the thermostat. The noise vulnerability of a Schmitt triggered input pin is lower. It bears Otto H. Schmitt's name since he created it in 1934 when he was a doctoral student researching the neural impulse transmission in squid nerves. Introduction to Embedded Systems, Lee & Seshia A single electrical connection may be shared by several devices in numerous applications. The designer must exercise caution to prevent a short circuit that might result in overheating and device failure if various devices concurrently push the voltage of this single electrical connection to different levels.

Suppose a manufacturing floor where a number of separate microcontrollers may all shut off a piece of equipment by asserting a logical zero on an output GPIO line. Since the microcontrollers may be redundant, a failure of one does not preclude a safety-related shutdown from happening, such a design may provide extra safety. We must take steps to make sure that the microcontrollers do not short each other out if all of these GPIO lines are connected to a single control input of the piece of equipment. This would happen if one microcontroller tried to drive a shared line to a high voltage while another tried to drive a low voltage on the same line.

Open collector circuits may be used for GPIO outputs. With such a circuit, turning on the transistor lowers the voltage on the output pin to (near) zero by putting a logical one into the memory mapped register. The output pin is left "open" when the transistor is turned off by writing a logical zero into the register. The shared line is linked to a pull-up resistor, which raises the voltage of the line to VDD when all the transistors are off. Many open collector interfaces may be connected as illustrated. When a single transistor is switched on, the whole line's voltage will drop to (or almost reach zero without causing a short circuit with the other GPIO pins. Logically, the output must be high if all registers contain zeros. The output will be low if any one of the registers contains a one. With active high logic, a circuit like this is referred to as a wired NOR since it is performing the NOR logical operation. Wired OR and wired AND may be produced in a similar manner by altering the arrangement.

CONCLUSION

Input and output (I/O) hardware devices are essential components of modern computer systems, enabling users to interact with digital information and manipulate it in various ways. I/O devices come in many forms, such as keyboards, mice, monitors, printers, and speakers, and their integration into wearable technology and IOT devices has increased their availability and importance in daily life.

REFERENCES

[1] Z. Zhang, H. Li, Y. Dong, X. Wang, and X. Dai, “Decentralized Signal Detection via Expectation Propagation Algorithm for Uplink Massive MIMO Systems,” *IEEE Trans. Veh. Technol.*, 2020, doi: 10.1109/TVT.2020.3008151.

[2] D. Qin and Z. Ding, “Exploiting Multi-Antenna Non-Reciprocal Channels for Shared Secret Key Generation,” *IEEE Trans. Inf. Forensics Secur.*, 2016, doi: 10.1109/TIFS.2016.2594143.

[3] W. Q. Wang, “Large time-bandwidth product MIMO radar waveform design based on chirp rate diversity,” *IEEE Sens. J.*, 2015, doi: 10.1109/JSEN.2014.2360125.

[4] W. Liu, C. H. Chang, F. Zhang, and X. Lou, “Imperceptible misclassification attack on deep learning accelerator by glitch injection,” in *Proceedings - Design Automation Conference*, 2020. doi: 10.1109/DAC18072.2020.9218577.

[5] Q. Lv, J. Li, P. Zhu, D. Wang, and X. You, “Downlink spectral efficiency analysis in distributed massive mimo with phase noise,” *Electron.*, 2018, doi: 10.3390/electronics7110317.

[6] T. K. Vu and Q. L. Tran, “Robust Loss Functions: Defense Mechanisms for Deep Architectures,” in *Proceedings of 2018 10th International Conference on Knowledge and Systems Engineering, KSE 2018*, 2018. doi: 10.1109/KSE.2018.8573318.

[7] S. E. E. Alex, “Challenging computer-based projects for a mechatronics course: Teaching and learning through projects employing virtual instrumentation,” *Comput. Appl. Eng. Educ.*, 2006, doi: 10.1002/cae.20083.

[8] R. M. Hussein, “Analysis/design of multistory concrete buildings on microcomputers,” *Adv. Eng. Softw. Work.*, 1991, doi: 10.1016/0961-3552(91)90042-3.

CHAPTER 13

MULTITASKING PROCESS AND MESSAGE PASSING

Mr. Aishwary Awasthi, Research Scholar
Department of Mechanical Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- aishwary@sanskriti.edu.in

ABSTRACT:

Multitasking refers to the ability of an embedded system to execute multiple tasks simultaneously. This is achieved by dividing the available processing resources into smaller tasks, which can be executed in parallel. Multitasking allows embedded systems to handle multiple tasks, such as data acquisition, data processing, and user interface management, at the same time. Multitasking can be implemented in embedded systems through preemptive scheduling or cooperative scheduling. Preemptive scheduling involves the operating system interrupting the running task to allow another task to execute. Cooperative scheduling involves tasks voluntarily giving up the processor to allow other tasks to execute.

KEYWORDS:

Cooperative Scheduling, Data Processing, Embedded System, Interface Management, Operating System.

INTRODUCTION

Embedded systems are computer systems that are designed to perform a specific set of functions. They are designed to work in real-time with limited hardware resources. In embedded systems, multitasking and message passing are two important concepts that are widely used. Multitasking is the ability of an operating system to run multiple tasks concurrently, whereas message passing is a communication mechanism that allows tasks to communicate with each other.

Multitasking:

Multitasking is an important feature of an operating system that allows multiple tasks to run concurrently. In an embedded system, there are multiple tasks that need to be executed simultaneously. These tasks may include communication, data acquisition, data processing, and control [1], [2]. To achieve multitasking in an embedded system, an operating system with a real-time kernel is used. A real-time kernel is a software component that provides scheduling, synchronization, and communication services to the tasks running on the system.

Scheduling:

Scheduling is the process of allocating the processor time to different tasks. In an embedded system, scheduling is critical as it determines the responsiveness and performance of the system. There are different scheduling algorithms that can be used in an embedded system. Some of the commonly used scheduling algorithms are Round Robin, Priority-based scheduling, and Rate Monotonic scheduling. Round Robin is a simple scheduling algorithm that assigns equal time slices to each task [3], [4]. Priority-based scheduling assigns priorities to tasks based on their importance, and the task with the highest priority is executed first. Rate Monotonic scheduling assigns priorities based on the task's execution time. Tasks with a shorter execution time are assigned higher priorities.

Synchronization:

Synchronization is the process of coordinating the activities of different tasks. In an embedded system, tasks may need to share resources such as memory, I/O devices, and CPU time. To avoid conflicts and ensure data consistency, synchronization mechanisms are used. Some of the commonly used synchronization mechanisms are semaphores, mutexes, and monitors. Semaphores are used to coordinate access to shared resources by limiting the number of tasks that can access the resource simultaneously. Mutexes are used to ensure that only one task at a time can access a shared resource. Monitors are used to provide mutual exclusion and condition synchronization between tasks.

Communication:

Communication is the process of exchanging information between tasks. In an embedded system, tasks may need to communicate with each other to coordinate their activities. There are different communication mechanisms that can be used in an embedded system [5], [6]. Some of the commonly used communication mechanisms are message passing, shared memory, and pipes. Message passing is a communication mechanism that allows tasks to communicate with each other by sending and receiving messages. Shared memory is a communication mechanism that allows tasks to share a common area of memory. Pipes are a communication mechanism that allows tasks to communicate by sending data through a pipe.

Message Passing:

Message passing is a communication mechanism that allows tasks to communicate with each other by sending and receiving messages. In an embedded system, message passing is commonly used as a communication mechanism. Message passing allows tasks to communicate with each other without sharing memory. This makes the system more robust and less prone to errors. There are different types of message passing mechanisms that can be used in an embedded system. Some of the commonly used message passing mechanisms are Direct Message Passing, Indirect Message Passing, and Mailboxes.

Direct Message Passing:

Direct Message Passing is a message passing mechanism that allows tasks to communicate with each other by sending messages directly. In Direct Message Passing, the sender task sends a message directly to the receiver task. The receiver task waits for the message and processes it when it arrives. Direct Message Passing is a simple and efficient communication mechanism [7], [8]. Embedded systems are designed to perform specific tasks in a constrained environment, and as such, multitasking is an essential requirement for most embedded systems. Multitasking in embedded systems refers to the ability of an embedded system to handle multiple tasks or processes simultaneously. In this context, tasks refer to the individual units of work that make up a system, and processes refer to the execution of those tasks.

DISCUSSION

In general, there are two primary methods for implementing multitasking in embedded systems: preemptive multitasking and cooperative multitasking. Preemptive multitasking involves the use of a scheduler that interrupts running tasks at predefined intervals, allowing other tasks to run. Cooperative multitasking, on the other hand, relies on tasks voluntarily giving up control of the system when they are finished executing.

Regardless of the approach taken, message passing is a fundamental concept in multitasking embedded systems. Message passing is the process of transmitting information between tasks or processes in a system. This communication can be accomplished through shared memory, message queues, or other inter-process communication (IPC) mechanisms. We will explore both multitasking and message passing in greater detail, including the benefits and challenges of each approach and best practices for implementing them in embedded systems.

Multitasking in Embedded Systems

Multitasking in embedded systems allows the system to perform multiple tasks simultaneously, making it more efficient and flexible. The ability to perform multiple tasks at once is especially important in real-time systems, where multiple processes must be executed in a timely and predictable manner.

In preemptive multitasking, the scheduler is responsible for managing the execution of tasks in the system. The scheduler interrupts running tasks at predefined intervals and assigns the CPU to another task. This approach ensures that all tasks in the system receive a fair share of the available processing time.

Cooperative multitasking, on the other hand, relies on tasks voluntarily giving up control of the system when they are finished executing. This approach requires careful coordination between tasks to ensure that they do not monopolize the system's resources.

Both approaches have their advantages and disadvantages, and the choice between them depends on the specific requirements of the embedded system.

Benefits of Multitasking in Embedded Systems

Multitasking in embedded systems provides several benefits, including:

1. **Increased Efficiency:** Multitasking allows the system to perform multiple tasks simultaneously, making it more efficient.
2. **Improved Flexibility:** Multitasking allows the system to adapt to changing requirements by switching between tasks as needed.
3. **Better Resource Utilization:** Multitasking ensures that all tasks in the system receive a fair share of the available processing time, making better use of the system's resources.
4. **Enhanced Real-Time Performance:** Multitasking is essential for real-time systems, where multiple processes must be executed in a timely and predictable manner.

Challenges of Multitasking in Embedded Systems

Multitasking in embedded systems also presents several challenges, including:

1. **Increased Complexity:** Multitasking adds complexity to the system, making it more difficult to design, implement, and test.
2. **Increased Overhead:** Multitasking requires additional overhead in terms of context switching and task scheduling, which can impact performance.
3. **Synchronization Issues:** Multitasking can lead to synchronization issues when tasks access shared resources concurrently.
4. **Increased Power Consumption:** Multitasking can lead to increased power consumption, as the system must maintain multiple tasks in a ready state.

We go through mid-level software methods that allow sequential code to run concurrently. There are several justifications for running multiple sequential processes simultaneously, but they are all time-related. By eliminating scenarios where long-running programmes might prevent a programme from responding to outside stimuli, such as sensor data or a user request, responsiveness can be improved. Reduced latency, or the interval between the appearance of a stimulus and the reaction, is a result of improved responsiveness. Increasing performance by enabling a programme to run in simulation is another argument simultaneously on a number of cores or CPUs. As it is assumed that it is best to do work sooner rather than later, this is also a time problem. To directly manage the time of external exchanges is a third justification. No of what other processes may be running at that moment, a programmed might need to do some action, like refreshing a display, at certain periods.

Concurrency has previously been covered in a number of scenarios. The link between the topic of this chapter and those of preceding which depicts how hardware offers concurrent methods to the software designer. The top layer, which includes abstract concurrency models like synchronous composition, dataflow, and time-triggered models. These two levels are connected in this chapter. It discusses mechanisms that may serve as infrastructure for the realisation of high-level mechanisms and are accomplished utilising low-level mechanisms. Multitasking, which refers to the simultaneous execution of numerous tasks, is the collective name for these intermediate approaches.

The high-level mechanisms are increasingly being used in place of the mid-level mechanisms by embedded system designers, who nevertheless commonly employ them directly to create applications. Using a software tool that supports a model of computation, the designer creates a model (or several models of computation). The model is then converted into a programme automatically or semi-automatically using the mid-level or lines of code as a degree of detail, however there is no guarantee that a line of C code runs atomically (it usually does not). In Figure 1 shows the multi-tasking and multi-threading differences.

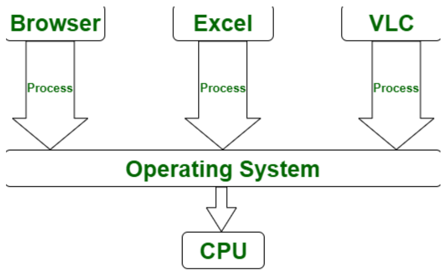


Figure 1: Illustrate the difference between Multi-tasking and Multi-Threading.

Nevertheless, realistic representations of C programmed often do not use finite state machines. A finite-state model is inappropriate when merely looking at the code, since it allows for the addition of any quantity of listeners to the list. The system has a limited state if and main process are combined since only three listeners are added to the list. So, the whole programmed would have to be included in an exact finite-state model, making it exceedingly challenging to reason about the code in a modular way.

When concurrency is included, the issues get substantially worse. In this chapter, we will demonstrate how difficult and error-prone it is to accurately reason about C programmes that use mid-level concurrency technologies like threads. Designers are gravitating towards the top layer . Imperative applications that run simultaneously and share memory are called threads. They are able to access one another's variables. Here, we shall use the word "threads" generally to refer to any method where imperative programmes run simultaneously and share memory. Several practitioners in the area use the term "threads" more strictly to refer to specific techniques of creating programmes that share memory. In this broad sense, threads are present on almost every microprocessor in the form of interrupts, even on hardware without an operating system.

In order to implement imperative applications that share memory, the majority of operating systems provide a higher-level mechanism than interrupts. A group of processes that a programmer may employ are offered as the mechanism. These processes often follow a defined API (application programme interface), which enables programmers to create portable (multiple processor and/or operating system) applications. Such such API is Pthreads (also known as POSIX threads), which is present in many contemporary operating systems. A selection of C programming language types, functions, and constants are defined by Pthreads. The IEEE standardised it in 1988 to bring all Unix variations under one umbrella.

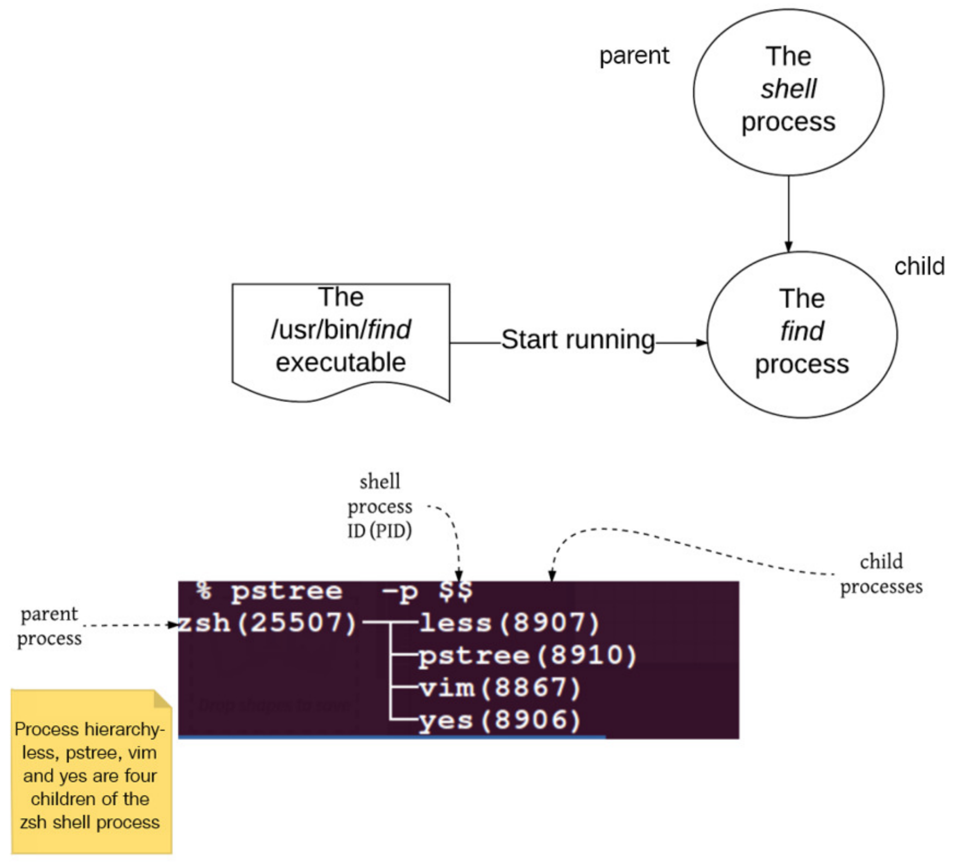
The key component of a thread implementation in Pthreads is a scheduler, which selects the thread to run next whenever a processor is free. The choice may be made in accordance with the fairness principle, which states that every active thread should be given an equal chance to run, time considerations, or any other metric of significance or priority scheduling methods are covered in great depth. Without any concern for how a thread scheduler chooses which thread to run, we only explain how it operates in this section.

When and how the scheduler is activated is the first important consideration. A thread is not interrupted by the straightforward cooperative multitasking approach unless the thread itself invokes a specific procedure or one of a certain set of processes. The scheduler, for instance, may step in anytime any operating system service is called by the thread that is now running. A library procedure call is used to activate an operating system service.

The return address will be placed into the stack when the procedure is called, and each thread has its own stack. The requested service is provided, and the operation resumes as usual, if the scheduler decides that the thread that is presently running should continue to run. Instead, if the scheduler decides that the thread should be suspended and a different thread should be chosen to execute, it does not just return but instead records the stack pointer of the thread that is now running and changes it to refer to the selected thread's stack. By popping the return address off the stack and continuing execution in a separate thread, it then returns as usual.

The fundamental drawback of cooperative multitasking is that other threads may run for an extended period of time without any operating system service requests, starving them of resources. Most operating systems provide an interrupt service procedure that executes at predetermined times to remedy for this. This procedure will keep track of the system clock, enabling periodic scheduler invocation through a timer interrupt and giving application programmers a mechanism to find out the time of day. A jiffy is the time interval at which the system-clock ISR is called in an operating system with a system clock.

It might be difficult to prevent deadlock. Deadlock may be avoided by removing any of the requirements Coffman et al. (1971) list as essential for it to happen. One straightforward method is to employ a single lock across a multithreaded application. Nevertheless, this method does not provide highly modular code. Moreover, it might make it challenging to adhere to real-time restrictions since certain shared resources such as displays might need to be retained for an extended period of time, delaying deadlines in other threads. In Figure 2 shows the message passing models.


parent
The *shell* process
child
The /usr/bin/*find* executable
Start running
The *find* process
shell process ID (PID)
child processes
parent process
Process hierarchy- less, pstree, vim and yes are four children of the zsh shell process

```
% pstree  -p $$
zsh(25507)─┬─less(8907)
           ├─pstree(8910)
           ├─vim(8867)
           └─yes(8906)
```

Figure 2: Illustrate the message passing model.

We can sometimes utilise the interrupts' ability to be enabled and disabled as a single global mutex in a very basic microkernel. Suppose that there is just one processor (not a multicore) and that a thread may only be suspended through an interrupt; it cannot be suspended by using kernel services or blocking I/O. Disabling interrupts prevents a thread from being suspended under certain presumptions. This method won't work in the majority of OSs since threads may be halted for a variety of reasons.

A third method is to make sure that each thread receives the locks in the same sequence when there are multiple mutex locks. However for a number of factors, it might be difficult to ensure. First off, because most programmes are built by a team of authors, a procedure's locks are not included in the procedure's signature. As a result, this method depends on thorough documentation that is consistent throughout, as well as collaboration between Lee & Seshia, Introduction to Embedded Systems 307 11. 2. A development team THREADS. Every time a lock is introduced, the program's components that acquire locks may need to be changed.

Second, it may make accurate coding very difficult. Programmers must first release any locks they are holding before calling a method that obtains lock1, which is always the first lock obtained in a programme by convention. When it releases those locks, it might be halted and the resource they were protecting could change. Once it obtains lock 1, it must then reacquire those locks, but it will need to presume that it has lost all knowledge of the condition of the resources and may need to perform a significant amount of work.

CONCLUSION

Multitasking and message passing are important techniques in modern computer systems. Multitasking allows us to run multiple tasks or programs concurrently, improving the responsiveness and efficiency of our systems. Message passing allows different tasks or processes to communicate with each other, enabling them to work together and share information. Together, these techniques are essential for building complex, responsive, and efficient computer systems.

REFERENCES

[1] S. Verstocken, D. Van De Putte, P. Camps, and M. Baes, “SKIRT: Hybrid parallelization of radiative transfer simulations,” *Astron. Comput.*, 2017, doi: 10.1016/j.ascom.2017.05.003.

[2] W. R. Bevier, “Kit: A Study in Operating System Verification,” *IEEE Transactions on Software Engineering*. 1989. doi: 10.1109/32.41331.

[3] N. Wang, H. Y. Chen, Y. W. Chen, and Y. C. Lin, “The benefits of concurrent computing in tribology system design,” *Tribol. Int.*, 2019, doi: 10.1016/j.triboint.2018.12.008.

[4] W. Harrison and A. Procter, “Cheap (but functional) threads,” *Submitt. to J. Funct. Program.*, 2005.

[5] E. F. V. Martins and A. Nunes Da Cruz, “An operating system extension for a multiprocessor,” *J. Syst. Archit.*, 1998, doi: 10.1016/S1383-7621(98)00002-2.

[6] M. TAYA and R. J. ARSENAULT, “CHAPTER 1 - Introduction,” *Met. Matrix Compos.*, 1989.

[7] C. Katsinis and P. Hogan, “Concurrent multitasking applications on a shared-bus multicomputer system,” *Comput. Syst. Sci. Eng.*, 1996.

[8] H. Gill *et al.*, “Integrating knowledge-based technology into computer aided ventilation systems,” *Int. J. Clin. Monit. Comput.*, 1990, doi: 10.1007/BF02915525.

CHAPTER 14

SCHEDULING NETWORK BASED METHOD AND MUTUAL EXCLUSION

Mr. Aishwary Awasthi, Research Scholar
Department of Mechanical Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- aishwary@sanskriti.edu.in

ABSTRACT:

Scheduling refers to the process of determining which task or process to execute next on a shared resource, such as a CPU or a network connection. Scheduling algorithms can vary in complexity and can be optimized based on various factors, including task priority, time-slicing, and load balancing. Effective scheduling is critical for ensuring that tasks are executed efficiently and with minimal delay, which can improve the overall performance of a system. Mutual exclusion is a technique used to prevent multiple processes from accessing a shared resource simultaneously. This is necessary to avoid synchronization issues that can occur when multiple processes try to modify the same data concurrently. Mutual exclusion can be implemented using locks, semaphores, and other mechanisms. Effective mutual exclusion ensures that processes can safely access shared resources without interference or data corruption.

KEYWORDS:

Algorithm, Data corruption, Network Connection, Scheduling, Mutual Exclusion.

INTRODUCTION

Scheduling and mutual exclusion are two fundamental concepts in computer science that are essential for developing efficient and reliable systems. Scheduling refers to the process of allocating resources to different tasks or processes, while mutual exclusion refers to the prevention of two or more processes from accessing the same resource simultaneously [1], [2]. In this article, we will explore both concepts in greater detail, including their importance, benefits, challenges, and best practices. Scheduling is the process of allocating resources to different tasks or processes in a system. Resources can include CPU time, memory, input/output devices, and other system resources. Scheduling is essential for ensuring that a system operates efficiently and effectively.

The scheduling process involves several key steps, including:

1. **Prioritization:** Prioritizing tasks or processes based on their importance or urgency.
2. **Allocation:** Allocating system resources to tasks or processes based on their priority and available resources.
3. **Execution:** Executing tasks or processes in the allocated resources, based on their priority and available resources.

There are several types of scheduling algorithms, including:

1. **First-Come, First-Served (FCFS):** In this algorithm, the first task or process that arrives is the first to be executed.
2. **Round-Robin:** In this algorithm, each task or process is given a fixed amount of time to execute, and the CPU switches to the next task or process when the time is up.
3. **Priority-Based:** In this algorithm, tasks or processes are executed based on their priority level.
4. **Shortest Job First (SJF):** In this algorithm, the task or process with the shortest estimated execution time is executed first.

Benefits of Scheduling

Scheduling provides several benefits, including:

1. **Resource Allocation:** Scheduling ensures that system resources are allocated efficiently and effectively, which can improve the overall performance of the system.
2. **Time Management:** Scheduling enables the system to manage its time more effectively, which can help ensure that critical tasks are completed on time.
3. **Fairness:** Scheduling can help ensure that all tasks or processes in the system are given a fair share of system resources.

Challenges of Scheduling

Scheduling also presents several challenges, including:

1. **Complexity:** Scheduling can be a complex process, requiring careful coordination and synchronization of different tasks or processes.
2. **Overhead:** Scheduling can introduce overhead into the system, which can impact the overall performance of the system.
3. **Synchronization Issues:** Scheduling can lead to synchronization issues when tasks or processes access shared resources concurrently.

Best Practices for Scheduling

To ensure that scheduling in a system is implemented effectively, several best practices should be followed, including:

1. **Use Appropriate Scheduling Algorithms:** Different scheduling algorithms are suited to different types of systems and workloads. Choosing the appropriate algorithm can help ensure that system resources are allocated efficiently and effectively.
2. **Prioritize Critical Tasks:** Prioritizing critical tasks or processes can help ensure that they are given the necessary system resources to complete on time.
3. **Monitor System Performance:** Monitoring system performance can help identify bottlenecks or other issues that may be impacting the system's performance.

Mutual Exclusion

Mutual exclusion is the process of preventing two or more processes from accessing the same resource simultaneously. In computer science, resources can refer to shared memory, input/output devices, and other system resources.

Mutual exclusion is essential for ensuring the correct and consistent operation of a system. If two or more processes attempt to access the same resource simultaneously, it can lead to race conditions, deadlocks, or other synchronization issues [3], [4].

The most common approach to implementing mutual exclusion is through the use of locks. Locks are data structures that allow a process to request exclusive access to a resource. When a process holds a lock, it is the only process that can access the resource. Other processes must wait until the lock is released before they can access the resource.

There are several types of locks, including:

1. **Binary Semaphore:** A binary semaphore is a lock that has two states: locked and unlocked. Only one process can hold the lock at any given time.
2. **Mutex:** A mutex is a lock that allows only one process to access a resource at a time. If a process attempts to access the resource while the mutex is held by another process, it will block until the mutex is released.
3. **Spin Lock:** A spin lock is a lock that uses a busy-wait loop to request exclusive access to a resource. If the lock is already held by another process, the requesting process will spin, repeatedly checking the lock until it becomes available.

Benefits of Mutual Exclusion

Mutual exclusion provides several benefits, including:

1. **Correctness:** Mutual exclusion ensures that only one process can access a resource at a time, which can help prevent race conditions and other synchronization issues [5], [6].
2. **Performance:** Mutual exclusion can improve the performance of a system by allowing processes to execute concurrently, while still ensuring that they do not access the same resource simultaneously.
3. **Consistency:** Mutual exclusion can help ensure that a system operates consistently and reliably, even in the presence of multiple processes accessing the same resources.

Challenges of Mutual Exclusion

Mutual exclusion also presents several challenges, including:

1. **Deadlocks:** Deadlocks can occur when two or more processes are waiting for each other to release a lock, which can lead to a system freeze.
2. **Priority Inversion:** Priority inversion can occur when a high-priority process is blocked by a low-priority process that is holding a lock.
3. **Overhead:** Locking and unlocking resources can introduce overhead into the system, which can impact the overall performance of the system.

Best Practices for Mutual Exclusion

To ensure that mutual exclusion is implemented effectively in a system, several best practices should be followed, including:

1. **Use Appropriate Locking Mechanisms:** Different locking mechanisms are suited to different types of resources and workloads. Choosing the appropriate locking mechanism can help ensure that mutual exclusion is implemented effectively [7], [8].
2. **Avoid Deadlocks:** Deadlocks can be avoided by ensuring that processes release locks in the order in which they were acquired.
3. **Use Priority Inheritance:** Priority inheritance can be used to prevent priority inversion by temporarily raising the priority of a low-priority process that is holding a lock that a high-priority process needs.

DISCUSSION

Scheduling and mutual exclusion are two fundamental concepts in operating systems that are critical for ensuring the correct and efficient functioning of a computer system. Scheduling refers to the process of allocating resources such as processor time, memory, and input/output devices to multiple processes or threads running concurrently. On the other hand, mutual exclusion refers to the process of ensuring that shared resources are accessed by only one process or thread at a time to prevent conflicts and ensure data consistency. In this article, we will explore these two concepts in detail.

Scheduling:

Scheduling is the process of allocating resources to different processes or threads in a computer system. The scheduler is a key component of the operating system that decides which process or thread should be executed at any given time. The scheduler uses various algorithms to make this decision, and the choice of algorithm depends on the specific requirements of the system. The majority of solutions, however, either place onerous restrictions on the programmer or demand a high level of skill, which raises the possibility that there is an issue with the concurrent programming model of threads.

Deadlock and race circumstances are troublesome enough without adding threads' additional, maybe subtle issues with the programmes' memory model. A memory consistency model is provided by each thread implementation, defining how variables that are read and written by various threads seem to those threads. The latest value entered into a variable should be returned when reading it, but what does "last" really mean? Imagine, for instance, that thread A runs the following two statements: 1 x = 1; 2 w = y; and thread B executes the following two statements:

1 y = 1; 2 z = x; all variables are initialised with a value of 0.

Intuitively, we would anticipate that at least one of the two variables w and z has a value of 1 after the execution of these lines by both threads. Sequential consistency is the name given to this kind of assurance (Lamport, 1979). The concept of sequential consistency states that the outcome of any execution is the same as if all threads' activities were carried out in some sequential order. Seshia and Lee, Overview of Embedded Systems

Unfortunately, the majority (or even all) of Pthreads implementations do not ensure sequential consistency. In actuality, delivering such a guarantee on contemporary processors with contemporary compilers is pretty challenging. Since there is no dependence between them (that the compiler can see), it is possible for a compiler to rearrange the instructions in each of these threads. The hardware may reorganise them even if the compiler doesn't. An effective defensive strategy is to use mutual exclusion locks to very carefully restrict such accesses to shared variables (while hoping that the mutual exclusion locks themselves are implemented appropriately). In Figure 1 illustrate the difference between mutual exclusion vs asynchronous scheduling.

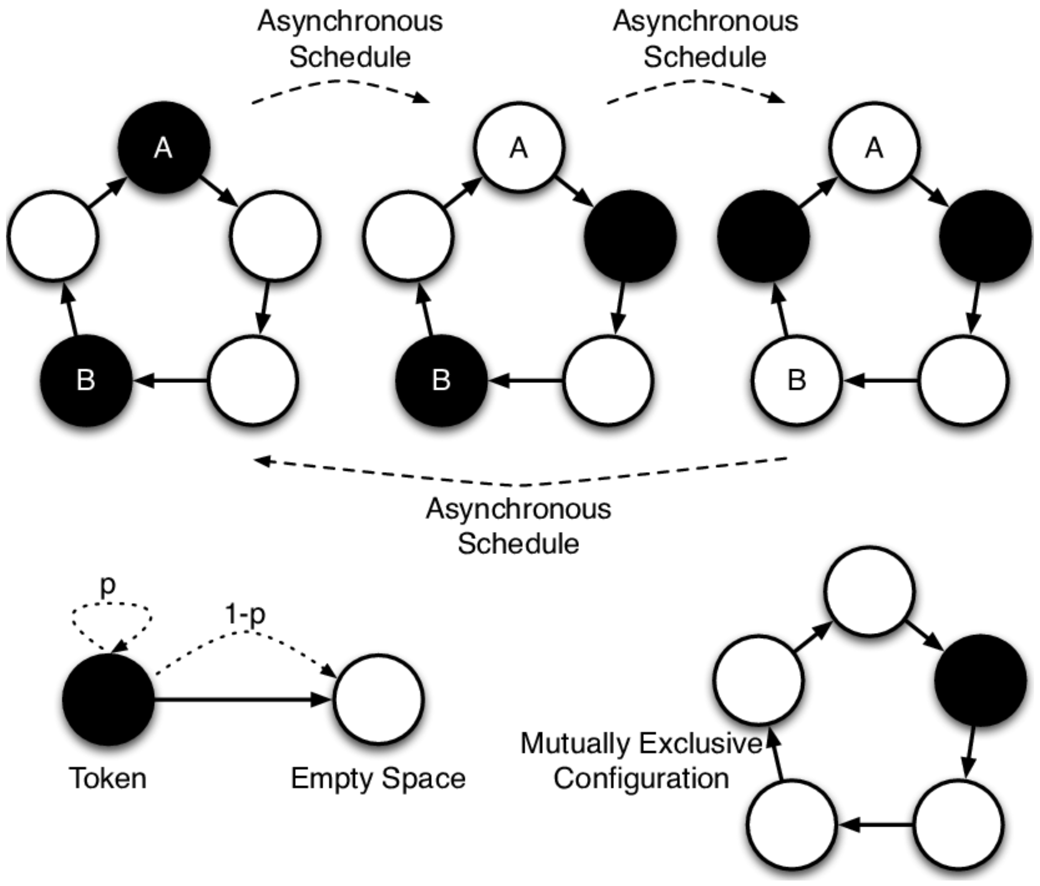


Figure 1: Illustrate the Mutual exclusion vs. asynchronous scheduling.

Multithreaded applications may be quite challenging to comprehend. Moreover, since bugs in the code may not surface during testing, it might be challenging to increase trust in the systems. For instance, a software may be susceptible to deadlock yet still function well for years without the problem. While programmers must exercise extreme caution, it is possible that programming mistakes will continue since it is so difficult to reason about the programmes. A straightforward trick allows us to avoid the potential deadlock, but the trick results in a more pernicious error an error that may not be discovered during testing and may not be noticed when it is discovered, unlike a deadlock, which is almost always discovered when it is discovered. There are different types of scheduling algorithms that are commonly used in operating systems. Some of these algorithms are:

1. **First-Come, First-Served (FCFS) Scheduling:**

In FCFS scheduling, processes or threads are executed in the order in which they arrive. The first process or thread that arrives is executed first, followed by the second process or thread, and so on. This algorithm is simple and easy to implement, but it can lead to long waiting times for processes or threads that arrive later.

2. **Shortest Job First (SJF) Scheduling:**

In SJF scheduling, the process or thread with the shortest expected processing time is executed first. This algorithm is useful in reducing the average waiting time for processes or threads, but it requires accurate estimates of processing time, which can be difficult to obtain.

3. **Priority Scheduling:**

In priority scheduling, each process or thread is assigned a priority, and the process or thread with the highest priority is executed first. This algorithm can be useful in ensuring that critical processes or threads are executed first, but it can also lead to starvation, where low-priority processes or threads never get a chance to execute.

4. **Round-Robin (RR) Scheduling:**

In RR scheduling, each process or thread is assigned a time slice, and the scheduler switches between processes or threads when their time slice expires. This algorithm is useful in ensuring that all processes or threads get a fair share of the processor, but it can lead to high overhead if the time slice is too small.

Mutual Exclusion:

Mutual exclusion is a mechanism for ensuring that shared resources are accessed by only one process or thread at a time. Shared resources can include variables, data structures, and input/output devices. Without mutual exclusion, multiple processes or threads can access shared resources simultaneously, which can lead to conflicts and inconsistencies in the data. There are different mechanisms for implementing mutual exclusion in a computer system. Some of these mechanisms are:

1. **Locks:**

Locks are a simple mechanism for implementing mutual exclusion. A lock is a variable that is used to indicate whether a resource is currently being accessed by a process or thread. When a process or thread wants to access a resource, it must first acquire the lock. If the lock is already held by another process or thread, the requesting process or thread must wait until the lock is released.

2. **Semaphores:**

Semaphores are a more complex mechanism for implementing mutual exclusion. A semaphore is a variable that is used to control access to a shared resource. A semaphore can have two values: 0 and 1. When a process or thread wants to access a resource, it first tries to decrement the semaphore value. If the semaphore value is already 0, the requesting process or thread must wait until the semaphore value becomes 1.

3. **Monitors:**

Monitors are a higher-level mechanism for implementing mutual exclusion. A monitor is a data structure that encapsulates shared resources and the operations that can be performed on them. Only one process or thread can access a monitor at a time, and all operations on the monitor are atomic, meaning that they are executed as a single, indivisible unit.

4. **Reader-Writer Locks:**

Reader-writer locks are a type of lock that is used when multiple processes or threads need to read from a shared resource, but only one process or thread can write to the resource at a time. The lock allows multiple processes or threads to read from the resource simultaneously, but it ensures that only one process or thread can write to the resource at a time. In Figure 2 illustrate the mutual exclusion.

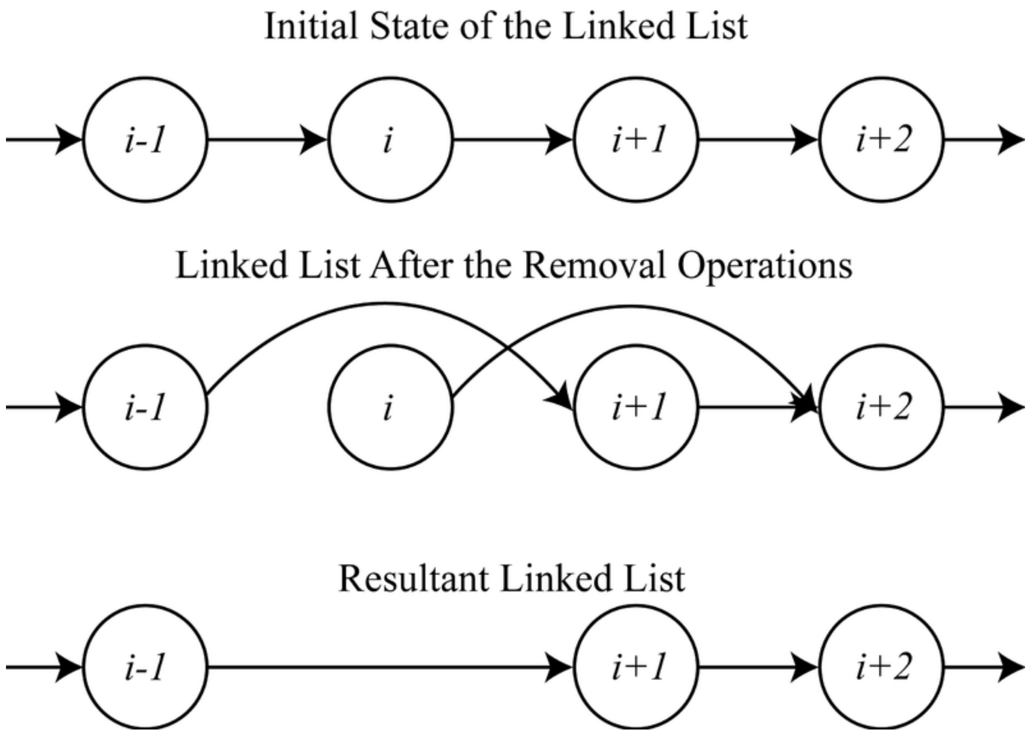

Initial State of the Linked List
i-1
i
i+1
i+2
Linked List After the Removal Operations
i-1
i
i+1
i+2
Resultant Linked List
i-1
i+1
i+2

Figure 2: Illustrate the mutual exclusion.

Deadlock:

Deadlock is a situation that can occur in a computer system when two or more processes or threads are waiting for each other to release a resource. This can lead to a situation where none of the processes or threads can proceed, resulting in a system freeze. Deadlock is a serious problem in operating systems, and various mechanisms have been developed to prevent or detect it.

Preemption:

Preemption is a mechanism that allows the scheduler to interrupt a running process or thread and allocate the processor to another process or thread. Preemption is used to ensure that critical processes or threads get a chance to execute, even if non-critical processes or threads are running. Preemption can also be used to prevent starvation, where low-priority processes or threads never get a chance to execute.

Scheduling and mutual exclusion are two fundamental concepts in operating systems that are critical for ensuring the correct and efficient functioning of a computer system. Scheduling refers to the process of allocating resources such as processor time, memory, and input/output devices to multiple processes or threads running concurrently. There are different types of scheduling algorithms that are commonly used in operating systems, and the choice of algorithm depends on the specific requirements of the system.

Mutual exclusion refers to the process of ensuring that shared resources are accessed by only one process or thread at a time to prevent conflicts and ensure data consistency. There are different mechanisms for implementing mutual exclusion in a computer system, including locks, semaphores, monitors, and reader-writer locks. Deadlock is a serious problem in operating systems, and various mechanisms have been developed to prevent or detect it. Preemption is a mechanism that allows the scheduler to interrupt a running process or thread and allocate the processor to another process or thread, and it is used to ensure that critical processes or threads get a chance to execute.

Real-time systems are collections of tasks with temporal limitations in addition to any ordering constraints imposed by task precedences. These limitations link a job's execution to real time, which is the actual time that exists in the setting where the computer is doing the work. Tasks often have deadlines, which are quantified amounts of time by which the job must be finished. Generally speaking, real-time

Programs may be subject to a variety of temporal restrictions in addition to deadlines. For instance, a job could need to start at a certain time and end no later than a certain point in the future, or it might need to finish no later than a certain length of time after another activity, or it might need to finish regularly every predetermined amount of time. Tasks may be interdependent and work together to produce an application. Instead, other than sharing CPU resources, they could not be connected. Each of these scenarios calls for a scheduling plan the various scheduling options, the characteristics of the tasks that a scheduler utilises to direct the process, and how schedulers are implemented in an operating system or microkernel.

CONCLUSION

Scheduling involves the allocation of shared resources, such as a CPU or I/O device, to multiple tasks or processes in a way that ensures efficient execution and minimal delay. Different scheduling algorithms can be used to optimize system performance, such as round-robin scheduling, priority-based scheduling, and load balancing. Mutual exclusion, on the other hand, is a technique that ensures that multiple processes cannot simultaneously access a shared resource. This is critical to avoiding synchronization issues that can result in race conditions and data corruption. Mutual exclusion can be implemented using locks, semaphores, and other mechanisms to ensure that processes can safely and effectively access shared resources.

REFERENCES

[1] B. S. Baker and E. G. Coffman, “Mutual exclusion scheduling,” *Theor. Comput. Sci.*, 1996, doi: 10.1016/0304-3975(96)00031-X.

[2] D. Kagaris and S. Dutta, “Scheduling mutual exclusion accesses in equal-length jobs,” *ACM Trans. Parallel Comput.*, 2019, doi: 10.1145/3342562.

[3] F. Gardi, “Mutual exclusion scheduling with interval graphs or related classes: Complexity and algorithms,” *4OR*, 2006, doi: 10.1007/s10288-005-0079-5.

[4] F. Gardi, “Mutual exclusion scheduling with interval graphs or related classes, Part I,” *Discret. Appl. Math.*, 2009, doi: 10.1016/j.dam.2008.04.016.

[5] G. Even, M. M. Halldórsson, L. Kaplan, and D. Ron, “Scheduling with conflicts: Online and offline algorithms,” *J. Sched.*, 2009, doi: 10.1007/s10951-008-0089-1.

[6] F. Gardi, “Mutual exclusion scheduling with interval graphs or related classes. Part II,” *Discret. Appl. Math.*, 2008, doi: 10.1016/j.dam.2007.08.017.

[7] W. Wu, J. Zhang, A. Luo, and J. Cao, “Distributed mutual exclusion algorithms for intersection traffic control,” *IEEE Trans. Parallel Distrib. Syst.*, 2015, doi: 10.1109/TPDS.2013.2297097.

[8] P. Li, H. Wu, B. Ravindran, and E. D. Jensen, “A utility accrual scheduling algorithm for real-time activities with mutual exclusion resource constraints,” *IEEE Trans. Comput.*, 2006, doi: 10.1109/TC.2006.47.

CHAPTER 15

A SYSTEMATIC APPROACH TO REACHABILITY ANALYSIS AND MODEL CHECKING

Sandeep Verma, Research Associate

Department of R&D Cell, IIMT University, Meerut Uttar Pradesh, India

Email id- sandeep91verma@gmail.com

ABSTRACT:

Reachability analysis involves determining whether a given state or configuration of a system can be reached from some initial state or configuration. This technique can be used to detect errors and verify the correctness of system behaviors, such as the absence of deadlocks and livelocks. Model checking, on the other hand, involves systematically exploring all possible states and behaviors of a system against a formal specification or model. This technique can be used to detect errors and verify the correctness of complex systems, including hardware and software designs.

KEYWORDS:

Complex System, Deadlock, Reachability Analysis, Model Checking.

INTRODUCTION

When presented with a decision about the execution of a concurrent programme or collection of programmes, a scheduler chooses which task to perform next. In general, a scheduler may have access to several processors for example in a multicore system. A multiprocessor scheduler must choose which work to carry out next, as well as on which processor. Processor assignment refers to the decision of which processor to use.

Any of these three choices may be made either at design time, just before the programme starts running, or at run time, while the programme is already running. There are a few distinct sorts of schedulers that we may categorise based on when the choices are made [1].All three choices are made at the design stage by a fully-static scheduler.

The end result of scheduling is a detailed description of what to do and when for each processor. Semaphores and locks are often not required by a fully-static scheduler. Instead, it may employ time to impose limitations on precedence and mutual exclusion. Nevertheless, since it is challenging to anticipate with accuracy how long a job will take to complete and because tasks often have data-dependent execution timings, fully-static schedulers are difficult to implement with the majority of current microprocessors.

A static order scheduler handles task assignment and ordering at design time, but the choice of when in physical time to execute a job is left to run time. This choice may be influenced, for instance, by the availability of a mutual exclusion lock or the fulfilment of precedence restrictions. With static order scheduling, before the programme starts running, each CPU is given its marching orders, and it simply carries them out as rapidly as it can. It does not, for instance, alter the sequence of tasks depending on the status of a lock or a semaphore.

Nevertheless, a task itself could latch onto a semaphore or a lock, in which case the subsequent tasks on that processor will all be blocked.

The assignment is completed at design time and the rest is done at run time via a static assignment scheduler. A run-time scheduler chooses which job to perform next while tasks are being executed; each processor is assigned a list of tasks to complete [2]. All actions are taken by a completely dynamic scheduler at run time. The scheduler decides what job to run next on a processor when it becomes available (e.g., when it completes an active task or when a task prevents obtaining a mutex). Of course, there are further scheduling options. For instance, a task's assignment may only be made once, at run time, shortly before the job is executed for the first time. The same assignment is utilised when running the same job again. Some combinations are not very logical. For instance, it makes no sense to decide the sequence of a job at run time and the time it will be executed at design time.

While a job is still being completed, a preemptive scheduler may decide to schedule another task for the same processor. In other words, a task may already be halfway through execution when the scheduler chooses to interrupt it and start. Preemption is the term for the first task being interrupted. A non-preemptive scheduler is one that never assigns another job to perform on the same processor before letting the current task complete [3], [4]. If a task tries to get a mutual exclusion lock during preemptive scheduling and the lock is not accessible, the job may be preempted. The job is considered to be blocked on the lock when this happens [5]. The stalled job may continue after another task releases the lock. A job may also be interrupted when it releases a lock. This may happen, for instance, if a process with a higher priority is impeded by a lock. In this chapter, we'll assume that all programmes are well-structured and that each job that gets a lock ultimately releases it.

A scheduler requires knowledge about the program's structure in order to make choices. The scheduler is often predicated on receiving a limited set T of tasks. Each job may or may not be taken into account as finite (ending in a limited amount of time). Real-time schedulers often presume that tasks will end, unlike the usual operating system scheduler. Several more presumptions regarding tasks may be made by a scheduler, some of which are covered in this section. The job model of the scheduler is the name of the collection of presumptions.

Some schedulers enable arrival of tasks, which means that tasks become known to the scheduler while other tasks are being completed, whereas other schedulers make the assumption that all tasks to be done are known before scheduling starts. Certain schedulers allow for circumstances in which each task T runs repeatedly, potentially indefinitely or just occasionally [6], [7]. A job may also be sporadic, which indicates that there is a lower constraint on the amount of time that must pass before it is executed again. We must distinguish between the task and the task executions 1, 2, in circumstances when a task T executes repeatedly. No such difference is required if each job runs precisely once. Priority constraints, or the need that one task execution come before another, may apply to task executions. If execution of I must occur before j, then I j may be used. In this case, tasks I and j may be separate executions of the same task or different tasks.

DISCUSSION

A task's execution may need specific prerequisites before it can begin or restart. These are prerequisites that must be met for the assignment to be completed. The task execution is said to be enabled when the prerequisites are met. For instance, precedences define the prerequisites before a task execution begins. A lock's accessibility could be a requirement before a job can be resumed. We define the release time ri (also known as the arrival time) as the earliest moment at which a task is enabled for an execution of a job i. The execution's real start time is what we refer to as the start time si. Naturally, si ri is something we must have. The moment the job has finished execution is the finish time fi, according to our definition. Therefore, fi si Oi = fi ri yields the reaction time oi.

Hence, the reaction time is the amount of time that passes between when a job is initially activated and when it has finished running. The entire amount of time the job is really running is defined as the execution time ei of i. That excludes any time the job could be delayed or interrupted. Several scheduling techniques use the erroneous assumption that a job will take 326 seconds to complete. Introduction to Embedded Systems, Lee & Seshia It is customary (and often erroneous) to assume that the worst-case execution time (WCET) is known when the execution time is changeable. Software execution time estimation may be fairly difficult, as explained in Chapter 16.

The time by which a job must be finished is the deadline di. In certain cases, a deadline is a genuine physical restriction placed by the application, and failing to meet the deadline is seen as a mistake. Hard deadlines are used to describe such deadlines. Hard real-time scheduling is defined as scheduling with strict deadlines. A deadline often indicates a design choice that need not be rigorously adhered to. Although it is preferable to meet the deadline, it is not a mistake to do so. It is often preferable to meet the deadline without much delay. Soft real-time scheduling is the situation in question.

Priority may be used instead of (or in addition to) a deadline by a scheduler. A priority-based scheduler works on the premise that each job has a priority allocated to it, and that it will always pick to carry out the task with the greatest priority (this is often represented by the item with the lowest priority number). A task's priority that is fixed applies to all executions of that task. Execution is able to modify the priority dynamically[8].

A scheduler that supports task arrivals and always is performing the enabled task with the greatest priority is known as a preemptive priority-based scheduler. A non-preemptive priority-based scheduler is one that never interrupts a job's execution to schedule another task; instead, it utilises priorities to decide which task to perform next once the current task execution has finished. In Figure 1 illustrate the Backward and forward reachability analysis for environment generation.

The objectives of the application have an impact on the decisions that are made about the scheduling approach. All job completions must fulfil their deadlines, fi di, which is a rather straightforward objective. A viable timetable is one that carries out this goal. A scheduler is considered to be optimum with regard to feasibility if it produces a viable schedule for each task set (that adheres to its task model) for which a feasible schedule exists.

The possible processor utilisation might be a metric to compare scheduling strategies. The utilisation is the proportion of time the processor is used to complete tasks (vs. being idle). For activities that run on a regular basis, this measure is most helpful. Evidently, the best scheduling algorithm in terms of practicality is one that produces a workable schedule whenever processor usage is less than or equal to 100%. The maximum lateness, Lmax, is another factor that might be used to evaluate schedulers. It is defined given a collection of task executions T as Lmax = max iT (fi di).

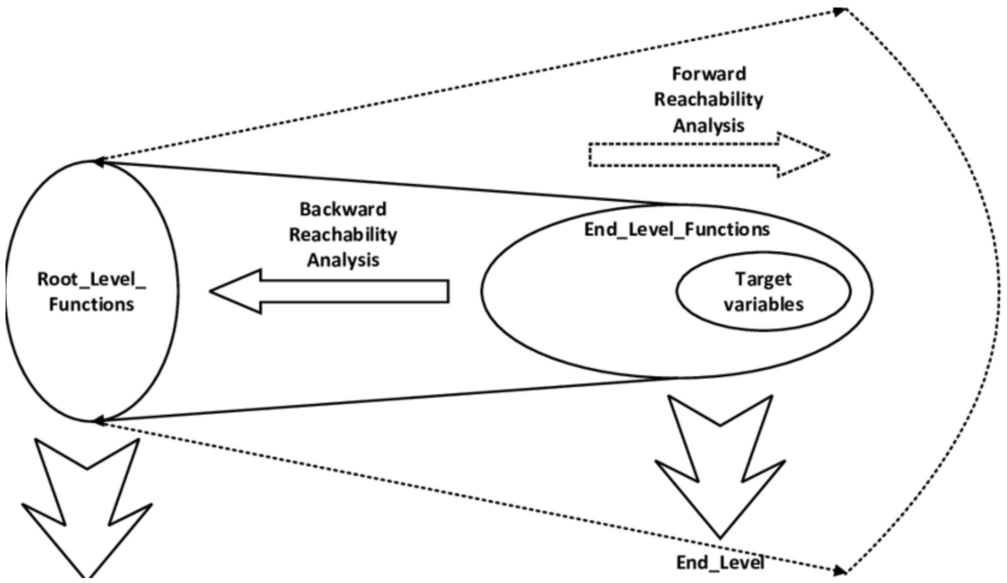


Figure 1: Illustrate Backward and forward reachability analysis for environment generation.

In the case of a workable timetable, its value is 0 or negative. Maximum lateness, however, may also be used to contrast unrealistic timetables. It may be OK for this number to be positive for soft real-time situations, provided that it does not exceed a certain threshold.

The total completion time or makespan, denoted by M = max iT fi min iT ri, is a third criteria that might be applied to a finite collection T of task executions.

If reducing the makespan is the scheduling objective, this is more of a performance objective than a real-time need. For scheduling choices made at run time, a scheduler may be a component of an operating system or microkernel, a part of a compiler or code generator, or both (if some scheduling decisions are made at design time and some at run time).

Tasks are often implemented by a run-time scheduler as threads (or processes, although the difference is not significant here). The scheduler may make the assumption that these threads finish in a certain amount of time or it may not. The scheduler is a process that is called at certain times in both scenarios. Every time a job is finished, a very basic, non-preemptive scheduler may call the scheduling process. When any of the following occurs for preemptive schedulers, the scheduling process is triggered:

a) An I/O interrupt occurs; a timer interrupt, for instance at a quick interval.
b) A system running service is called.
c) A task makes an effort to get a mutex.

In the other scenarios, the operating system process that delivers the service calls the scheduling procedure. In both situations, the necessary data to continue execution is present in the stack. The scheduler, however, may decide against just restarting execution.

In other words, it has the option to postpone returning from the interruption or service operation. Instead, it can decide to defer the ongoing job in favour of starting or continuing a different one. In Figure 2 illustrate the FSM analysis.

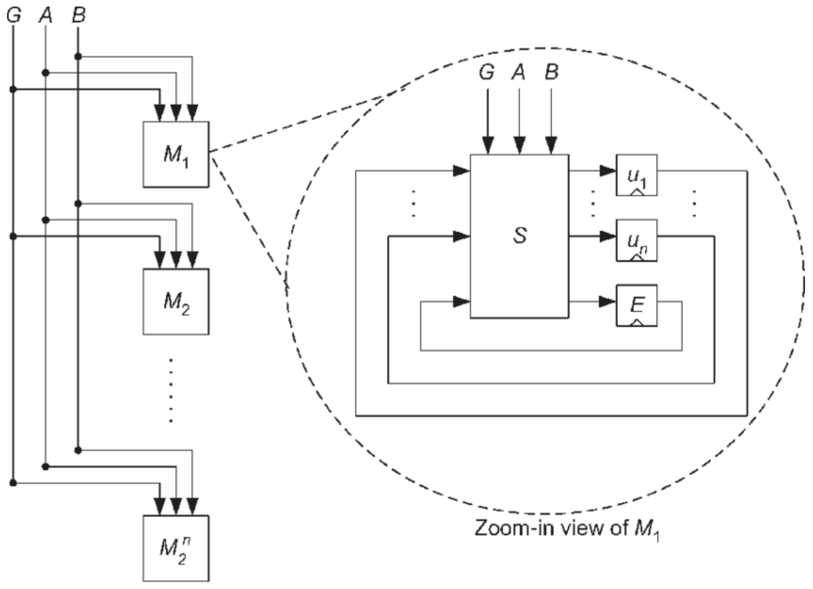


Figure 2: Illustrate the FSM for reachability analysis.

To carry out this preemption, the scheduler must log the preemption and, maybe, the reason for the preemption so that it can subsequently restart the job. The stack pointer may then be modified to reflect the status of the job that is about to be started or resumed. At that moment, a return is carried out, however this time, another job will be done instead of the task that was preempted.

It may be difficult to put a preemptive scheduler into practise. Concurrency has to be carefully controlled. For instance, it could be necessary to deactivate interrupts for a significant portion of the process in order to prevent a corrupted stack. Due to this, scheduling is one of a kernel's or microkernel's most essential duties. System stability and dependability are significantly impacted by the quality of the implementation.

Take an instance where T = 1, 2,..., n jobs, each of which must run at regular intervals. We specifically assume that every job I must complete itself precisely once throughout every time interval pi. We refer to pi as the task's period. As ri,1 is the release time of the first execution, jpi is the deadline for the j-th execution of i. With the job model mentioned above, Liu and Layland (1973) demonstrated that a straightforward preemptive scheduling method known as rate monotonic (RM) scheduling is the most feasible among fixed priority uniprocessor schedulers. A job with a shorter deadline is given more importance in this scheduling method.

As shown in Figure 12.2, the simplest version of the issue only includes two tasks, T = 1, 2, with execution times e1 and e2 and periods p1 and p2. In the picture, task 2's execution time (e2) is longer than task 1's period (p1). So, it is evident that a non-preemptive scheduler will not be used if these two jobs are to run concurrently on the same processor. A workable timetable for two periodic tasks T = 1, 2, with a preemptive scheduling that prioritises task 1 higher. If job 2 must run uninterruptedly through to completion, task 1 will sometimes miss deadlines.

Task 1 is assigned more importance in that figure since its duration is shorter. As a result, it runs whether or not 2 is running at the start of each time interval. If 2 is already running, 1 preempts it. The calculation is based on the assumption that it will take. This timetable is workable, as opposed to the schedule not being workable if priority number two had been given more priority.

It is simple to demonstrate that RM, out of all preemptive fixed priority schedulers, is the most feasible for the two task situation given the task model that is specified and the minimal context changeover time. As there are only two fixed priority schedules for this straightforward case—the RM schedule, which provides job 1 greater priority, and the non-RM schedule, which gives task 2 higher priority—it is simple to demonstrate. We just need to demonstrate that the RM plan is possible if the non-RM timetable is in order to demonstrate optimality. In reality, the notion that context changeover time is little is problematic. An abrupt context transition often results in significant cache-related delays on CPUs with caches. In addition, context change might result in significant operating system overhead.

Before to doing this, we must take into account the potential task execution alignments that may impact viability. As a lower priority job responds more slowly when its beginning phase coincides with a higher priority activity. In other words, the worst-case situation is when all tasks begin their cycles at once. Hence, we just need to think about this case. The non-RM schedule is workable in this worst-case situation, when release dates coincide, if and only if e1 + e2 p1. We need e2 p1 e1 so that task 2 allows enough time for 1 to complete before its deadline since task 1 is preempted by 2 in order for 1 to meet its deadline. e2 \sp2 \se1 \sp1 \sτ1 \sτ2

The RM schedule prioritises 1 more highly. The condition that e1 + e2 p1 is adequate, but not essential, for the RM schedule to be realisable. We only need to demonstrate that the RM plan is feasible if the non-RM timetable is possible in order to demonstrate that RM is best in terms of feasibility. Figure 12.6 makes it obvious that the RM timetable is viable if equation (12.1) is met. The RM plan is ideal in terms of practicality since it is one of only two schedules with two set priorities. The following theorem may be obtained by using the same proving method to any number of tasks (Liu and Layland, 1973):

Principle 12.1. If any priority assignment produces a feasible schedule under the conditions of a preemptive, fixed priority scheduler, a finite set of repeating tasks T = 1, 2,..., n, associated periods p1, p2,..., pn, and no precedence restrictions, then the rate monotonic priority assignment produces a feasible schedule. A timer interrupt with a time interval equal to the greatest common factor of the task periods makes it simple to create RM schedules. They may also be used in conjunction with several timer interruptions.

As it turns out, RM schedulers sometimes fail to reach 100% usage. RM schedulers in particular are restricted to having set priority. Due to this limitation, there are instances when a task set that produces a workable schedule is less than 100% used yet cannot accept any lengthening of execution times or shortening of periods. This indicates that there are idle processing cycles that cannot be utilised without endangering deadlines.

Thankfully, Liu and Layland (1973) demonstrate that this impact is limited. First, take notice that = Xn i=1 ei pi may be used to express the utilisation of n separate jobs with execution times ei and periods pi. The processor is always active if is equal to 1. So, it is obvious that if > 1 for any job set, that task set lacks a workable timetable. The RM schedule is feasible, as shown by Liu and Layland (1973), if is smaller than or equal to a utilisation constraint provided by n(21/n 1), (12.2).

Consider a few scenarios in order to comprehend this (very astonishing) finding. Initially, if n = 1 (there is only one job), then n(21/n 1) = 1; hence, the conclusion informs us that if utilisation is 100% or Lee & Seshia, Introduction to Embedded Systems 333 earliest deadline: 12.3 If FIRST is lower, the RM timetable is doable. This should be evident as there is just one job, = e1/p1, and it should be clear that the deadline can only be reached if e1 p1.

n(21/n 1) 0.828 for n = 2. Hence, the RM schedule will fulfil all deadlines if a task set with two tasks does not seek to consume more than 82.8% of the available processor time. The utilisation limit approaches ln(2) 0.693 as n grows in size. In other words, limn n(21/n 1) = ln(2) 0.693. This indicates that the RM schedule will fulfil all deadlines if a task set with any number of tasks does not seek to consume more than 69.3% of the available processor time. We loosen the fixed-priority restriction in the next section and demonstrate that dynamic priority schedulers can outperform fixed priority schedulers in terms of usage. The execution of the cost is a little more difficult.

The earliest due date (EDD), often known as Jackson's algorithm, is a straightforward scheduling technique that takes into account a limited number of non-repeating activities with deadlines and no precedence restrictions (Jackson, 1955). The EDD technique just carries out the jobs in the sequence of their due dates, starting with the item with the earliest deadline. The relative order of two jobs is irrelevant if they have the same due date. An EDD schedule is optimum in the sense that it minimises the greatest lateness, in comparison to all other potential orderings of the tasks, when given a finite collection of non-repeating tasks T = 1, 2, , n with associated deadlines d1, d2, , dn.

Proof. An exchange argument is all that is required to demonstrate this theorem. Take into account a random schedule that is not EDD. Due to the fact that such a timetable is not Acceptable, there must be two jobs, I and j, where I comes just after j, but dj comes before di. This illustration shows:

We can demonstrate that the maximum lateness of the new schedule is equal to or less than that of the previous timetable. The EDD schedule has been created if the aforementioned exchange is repeated until no more tasks are available for it. The EDD schedule has the smallest maximum lateness of any schedule since its maximum lateness is equal to that of the original schedule.

First, notice that if a job other than I or j determines the maximum lateness, then the two schedules have the same maximum lateness, and we are done demonstrating that the maximum lateness of the second schedule is not larger than that of the first schedule. If not, the first schedule's maximum lateness must be Lmax = max(fi di, fj dj) = fj dj, where the latter equivalence is evident from the image and arises from the facts that fi fj and dj di.

CONCLUSION

Reachability analysis involves determining whether a given state or configuration of a system can be reached from some initial state or configuration. This technique can be used to detect errors and ensure the correctness of system behaviors, including the absence of deadlocks and live locks. Model checking, on the other hand, involves systematically exploring all possible states and behaviors of a system against a formal specification or model. This technique can be used to detect errors and ensure the correctness of complex systems, including hardware and software designs.

REFERENCES

[1] J. Guan, Y. Feng, and M. Ying, "Decomposition of quantum Markov chains and its applications," *J. Comput. Syst. Sci.*, 2018, doi: 10.1016/j.jcss.2018.01.005.

[2] C. Rocha and C. Muñoz, "Synchronous set relations in rewriting logic," *Sci. Comput. Program.*, 2014, doi: 10.1016/j.scico.2013.07.008.

[3] D. Griffith, D. Cogan, E. Magidimisha, and R. Van Zyl, "Flight hardware verification and validation of the K-line fire sensor payload on ZACube-2," 2019. doi: 10.1117/12.2503094.

[4] S. Dutta, T. Kushner, S. Jha, S. Sankaranarayanan, N. Shankar, and A. Tiwari, "Sherlock : A Tool for Verification of Deep Neural Networks," *Vnn 19*, 2019.

[5] A. Boronat, R. Heckel, and J. Meseguer, "Rewriting logic semantics and verification of model transformations," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2009. doi: 10.1007/978-3-642-00593-0_2.

[6] B. K. Aichernig and M. Tappler, "Probabilistic black-box reachability checking (extended version)," *Form. Methods Syst. Des.*, 2019, doi: 10.1007/s10703-019-00333-0.

[7] V. Schuppan and A. Biere, "Efficient reduction of finite state model checking to reachability analysis," *Int. J. Softw. Tools Technol. Transf.*, 2004, doi: 10.1007/s10009-003-0121-x.

[8] B. K. Aichernig *et al.*, "Learning and statistical model checking of system response times," *Softw. Qual. J.*, 2019, doi: 10.1007/s11219-018-9432-8.

CHAPTER 16

PERSONALIZED QUANTITIVES ANALYSIS RECOMMENDATION

Mr. Aishwary Awasthi, Research Scholar
Department of Mechanical Engineering, Sanskriti University, Mathura, Uttar Pradesh, India
Email Id- aishwary@sanskriti.edu.in

ABSTRACT:

Quantitative analysis is a type of analytical method that uses mathematical and statistical models to assess and measure numerical data. This approach is used in various fields, including finance, economics, engineering, and social sciences, to understand and predict quantitative phenomena. In quantitative analysis, data is collected and analyzed using statistical methods and models to identify patterns, trends, and relationships. Various types of statistical techniques are employed, including regression analysis, hypothesis testing, and time series analysis, to evaluate the data and draw conclusions. These analyses can be used to make predictions, optimize systems, and make informed decisions.

KEYWORDS:

Analytical Method, Quantitive Analysis, Hypothesis Testing, Optimize System, Regression Analysis.

INTRODUCTION

EDD minimises the maximum lateness, making it the best option in terms of practicality. EDD does not, however, facilitate the arrival of tasks, and as a result, does not support the execution of tasks on a recurring or periodic basis. Thankfully, EDD may be simply modified to incorporate them, resulting in Horn's method or earliest deadline first (EDF). Any method that at any given moment executes the job with the earliest deadline among all received tasks is optimum with regard to reducing the maximum lateness. This is true for a collection of n independent tasks T = 1, 2,..., n with associated deadlines d1, d2,..., dn and arbitrary arrival timings. A similar interchange argument is used in the demonstration of this. Moreover, the result may be simply expanded to accommodate an infinite number of arrivals [1].

EDF is a dynamic priority scheduling method, it should be noted. A job may be given a different priority each time it is carried out if it is repeated. It may become harder to implement as a result. While it is possible to set alternative dates for tasks, the typical deadline for periodic tasks is the end of the task's period. EDF performs better than RM on average, while being more costly to execute (Buttazzo, 2005b). Secondly, only among fixed priority schedulers is RM ideal with regard to feasibility, while EDF is optimal with regard to feasibility among dynamic priority schedulers. EDF also reduces the maximum amount of lateness[2]–[4].

Moreover, EDF leads to fewer preemptions in real-world use, which reduces the expense associated with context switching. This often makes up for the implementation's additional complexity. Moreover, any EDF schedule with less than 100%, unlike RM, EDF is the best option for a work set without precedents since it minimises maximum lateness. What if there are examples already? Priorities between tasks may be expressed by a precedence graph given a limited collection of tasks.

Instance 12.1 Assume six tasks T = 1, 3, 5, and 6, each with an execution time of one. The task precedences. According to the diagram, job 1 must be completed before either task 2 or 3 may be completed, task 2 must be completed before either task 4 or 5, or task 3 must be completed before task 6. The graphic displays each task's due date. The EDF schedule is the one with that designation. The above example demonstrates that when there are precedents, EDF is not ideal [5], [6]. In 1973, Lawler (1973) presented a straightforward approach that reduces the maximum lateness and is optimum with precedences. The plan is quite straightforward. With a definite, limited collection of tasks and deadlines, Lawler's method builds the timetable backwards, starting with the item that must be completed last. The job with the earliest deadline is the one that must be completed last and on which no other assignment relies. The algorithm continues to build the schedule backwards, selecting the job with the newest deadline each time from among those whose dependents have already been scheduled. The resultant timetable for the preceding example, denoted LDF in Figure 12.7, is workable. Latest deadline first is the name of Lawler's algorithm (LDF).

LDF is best in terms of maximum lateness minimization, which also makes it best in terms of practicality. It does not, however, facilitate task arrival. EDF* (EDF with precedences), which encourages arrivals and reduces the maximum amount of tardiness. With this adjustment, we change the due dates for each assignment. Assume that T is the collection of all tasks [7], [8]. Let D(i) T be the collection of task executions in the precedence graph for a task execution I T that are directly dependent on i. We create a modified deadline d 0 I = min(di, min jD(i) (d 0 j ej)) for all executions I T. The crucial modification is that task 2's deadline has been lowered from 5 to 2, reflecting the reality that tasks after it have shorter deadlines. Due to this, job 2 is scheduled before task 3, creating a workable timetable. EDF* may be seen as a method for making deadlines more sensible. This method makes sure that the deadlines take into consideration the deadlines of succeeding jobs rather than accepting arbitrary deadlines as presented. It is not clear why job 2 in the example should have a deadline that is five days later than its predecessors. To fix this oddity, EDF* is used first, then EDF.

While the algorithms described thus far are theoretically straightforward, their actual results are anything from straightforward and often astound system designers [9], [10]. When tasks share resources and employ mutual exclusion to control access to those resources, this is especially true. In theory, a priority-based preemptive scheduler always has the high-priority enabled job running. Yet, it is possible for a job to be stalled while being executed while employing mutual exclusion. Serious issues may arise if the scheduling method does not take this possibility into consideration.

DISCUSSION

After a few days on the mission, the Pathfinder started irregularly missing deadlines, leading to complete system resets that each resulted in data loss. The issue was identified by engineers on the ground as priority inversion, in which a low priority meteorological job was holding a lock and obstructing a high priority task, while medium priority operations were carried out. In Figure 1 illustrate the quantity analysis.

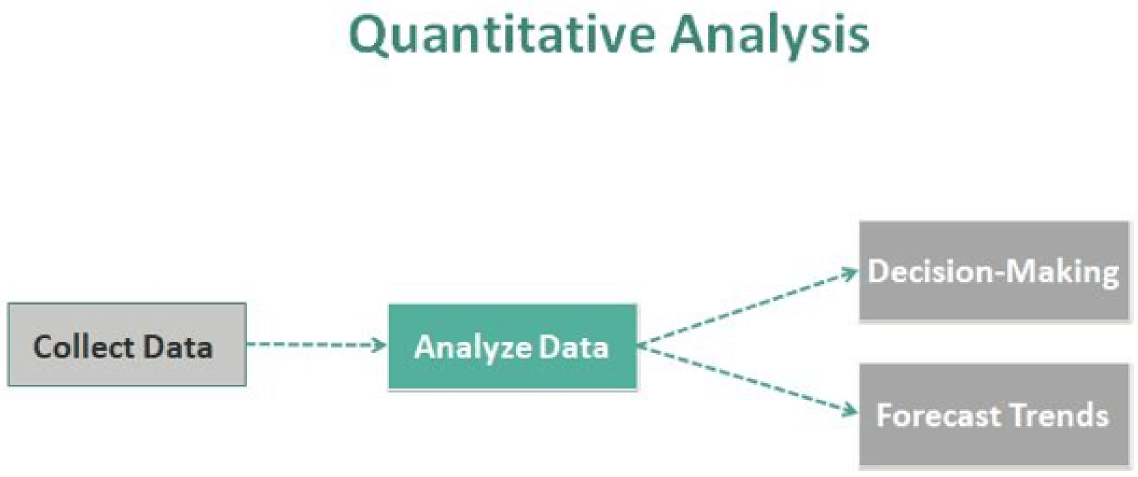


Figure 1: Illustrate the Quantity Analysis.

A high-priority activity being halted while unrelated, lower-priority processes are running is known as priority inversion. Task 3 in the diagram, which has low priority, gains a lock at time 1. It is preempted by task 1, a high-priority process, at time 2, and is subsequently blocked from attempting to get the same lock at time 3. Nevertheless, before task 3 gets to the point where it releases the lock, unrelated job 2, which has a medium priority, preempts it. Task 2 essentially stops the higher-priority task 1 from running since it can continue indefinitely. This almost likely isn't what you want.

Priority inheritance was introduced as a solution to the priority inversion issue in 1990 by Sha et al. According to their method, when a job blocks trying to acquire a lock, the task that already has the lock inherits the stalled task's priority. As a result, a job with a higher priority than the task seeking to obtain the lock cannot preempt the process that now has the lock. The priority order is task 1 highest, task 3 lowest. Task 3 enters a crucial part when obtaining a lock on a shared object. Task 1 preempts it, and when it attempts to acquire the lock, it is blocked. Task 1 is stalled for an infinite length of time because Task 2 preempts Task 3 at Time 4. As task 2 may keep task 1 waiting for whatever long it wants, the priorities of tasks 1 and 2 are effectively reversed. The priority order is task 1 highest, task 3 lowest. Task 3 enters a crucial part when obtaining a lock on a shared object. Task 1 preempts it, and when it attempts to acquire the lock, it is blocked. Task 3 inherits Task 1's priority, preventing Task 2 from preempting it.

Quantitative analysis is a complex field of study that is used to measure, analyze, and predict numerical data. This approach is applied in various disciplines, including finance, economics, engineering, social sciences, and medicine, to help professionals understand trends, patterns, and relationships that may be hidden in the data. In this essay, we will explore the basic concepts of quantitative analysis, its applications, and its limitations. Quantitative analysis involves the collection and analysis of numerical data, which is obtained through various methods, such as surveys, experiments, and simulations. The data is then analyzed using statistical techniques and models to identify patterns, trends, and relationships. The objective of quantitative analysis is to use the information obtained to make informed decisions, predict future events, and optimize systems.

In finance, quantitative analysis is used to identify market trends and predict the behavior of financial instruments. This is done using mathematical models and statistical techniques to analyze the historical performance of stocks, bonds, and other financial instruments. Quantitative analysts use this information to develop trading strategies, optimize investment portfolios, and manage risk. In economics, quantitative analysis is used to evaluate economic models, identify market trends, and predict consumer behavior. This is done using mathematical models and statistical techniques to analyze data related to economic indicators such as inflation, unemployment, and gross domestic product (GDP). These analyses help policymakers and business leaders make informed decisions about economic policies and investments.

In engineering, quantitative analysis is used to optimize systems and processes, identify design flaws, and predict the performance of products. This is done using mathematical models and simulations to analyze data related to the physical properties of materials, the behavior of mechanical systems, and the operation of industrial processes. In social sciences, quantitative analysis is used to study human behavior, identify trends, and predict outcomes. This is done using surveys and experiments to collect numerical data on variables such as demographics, attitudes, and behaviors. The data is then analyzed using statistical techniques and models to identify relationships and patterns. In medicine, quantitative analysis is used to study disease patterns, predict the effectiveness of treatments, and identify risk factors. This is done using clinical trials and observational studies to collect numerical data on health outcomes, such as mortality rates, morbidity rates, and quality of life. The data is then analyzed using statistical techniques and models to identify relationships and patterns.

The accuracy and reliability of quantitative analysis depend on the quality of the data collected and the validity of the underlying models used. Data must be collected and analyzed with care to ensure that it is representative of the population being studied and that it is not biased. Models must be designed and tested to ensure that they are accurate and can predict outcomes with a reasonable degree of certainty.

Quantitative analysis is not without its limitations, however. One of the main limitations is that it can only measure numerical data, which means that it cannot capture qualitative data such as emotions or opinions. Another limitation is that it assumes a linear relationship between variables, which may not always be the case in real-world situations. Additionally, quantitative analysis requires a high level of technical expertise, which may not be available in all contexts. To overcome these limitations, researchers may use a combination of quantitative and qualitative methods, such as interviews, focus groups, and surveys. This allows them to capture both numerical and qualitative data, providing a more comprehensive view of the phenomenon being studied.

Quantitative analysis can be divided into two main categories: descriptive statistics and inferential statistics. Descriptive statistics are used to summarize and describe numerical data using measures such as mean, median, and standard deviation. Inferential statistics are used to make predictions or draw conclusions about a population based on a sample of data. One important aspect of quantitative analysis is the use of statistical models to represent and analyze data. These models can be simple or complex, depending on the nature of the data being analyzed and the research questions being asked. Some of the most common statistical models used in quantitative analysis include regression analysis, time series analysis, and hypothesis testing.

Regression analysis is used to model the relationship between a dependent variable and one or more independent variables. For example, a regression model could be used to predict the price of a house based on its size, location, and other features. Time series analysis is used to model the behavior of data over time, such as stock prices or temperature readings. Hypothesis testing is used to determine the likelihood that a particular result is due to chance, and can be used to test whether a particular treatment or intervention is effective. Another important aspect of quantitative analysis is data visualization. Visualizing data can help researchers identify patterns and relationships that may be hidden in the data. Some of the most common types of data visualization include bar charts, scatterplots, and heat maps.

Quantitative analysis is also used in machine learning and artificial intelligence, where it is used to develop algorithms that can learn from and make predictions based on large datasets. This is used in a wide range of applications, from fraud detection in financial transactions to medical diagnosis. Figure 2 illustrate the application of quantitative data analysis in finance.

Consumer satisfaction survey
Quantitative analysis in finance
Data Projection
Application of quantitative data in finance
Annual income

Figure 2: Illustrate the application of quantitative data analysis in finance.

Quantity analysis is an approach used to measure and evaluate the quantity of items, goods, or services in various fields, such as economics, finance, and business. It involves the collection, processing, and interpretation of data to provide insights into the quantity of a particular item, which can then be used to make informed decisions.

I will discuss quantity analysis, including its definition, methods, applications, advantages, and limitations.

Definition of Quantity Analysis

Quantity analysis is the process of measuring and evaluating the quantity of a particular item, good, or service. It is a crucial aspect of data analysis in various fields, including economics, finance, and business. The primary aim of quantity analysis is to provide insights into the quantity of a particular item, which can then be used to make informed decisions.

Quantity analysis involves the collection of data through various methods, such as surveys, questionnaires, interviews, and secondary sources. Once the data is collected, it is processed and analyzed using statistical methods to obtain the desired information.

Methods of Quantity Analysis

There are various methods used in quantity analysis, depending on the type of data being analyzed. These methods can be broadly categorized into two groups: quantitative and qualitative methods.

Quantitative Methods

Quantitative methods involve the use of numerical data to analyze and evaluate the quantity of a particular item. These methods are more objective and precise than qualitative methods and involve the use of statistical techniques to analyze data.

Some common quantitative methods used in quantity analysis include:

1. **Survey Research:** This involves the use of questionnaires to collect data from a sample of people. The data is then analyzed using statistical methods to obtain the desired information.
2. **Regression Analysis:** This is a statistical method that involves the use of regression equations to analyze the relationship between two or more variables. It is used to predict the value of a dependent variable based on one or more independent variables.
3. **Time Series Analysis:** This involves the analysis of data collected over time to identify patterns and trends. It is used to predict future values of a particular item.
4. **Sampling:** This involves the selection of a small group of people or items from a larger population for analysis. The data obtained from the sample is then used to make inferences about the larger population.

Qualitative Methods

Qualitative methods involve the use of non-numerical data to analyze and evaluate the quantity of a particular item. These methods are more subjective and less precise than quantitative methods and involve the use of descriptive techniques to analyze data.

Some common qualitative methods used in quantity analysis include:

1. **Content Analysis:** This involves the analysis of the content of written or spoken communication to identify patterns and themes. It is used to evaluate the quantity of certain types of information.
2. **Case Studies:** This involves the in-depth analysis of a particular case to gain insights into a particular phenomenon. It is used to evaluate the quantity of certain characteristics or behaviors.
3. **Focus Groups:** This involves the use of group discussions to obtain information about a particular item. It is used to evaluate the quantity of certain opinions or beliefs.

Applications of Quantity Analysis

Quantity analysis has numerous applications in various fields, including economics, finance, and business. Some of the key applications of quantity analysis include:

1. **Economic Analysis:** Quantity analysis is used to measure the quantity of goods and services produced in an economy. This information is then used to make informed decisions about economic policies, such as setting interest rates or taxation levels.
2. **Financial Analysis:** Quantity analysis is used to evaluate the quantity of financial assets and liabilities held by individuals or organizations. This information is then used to make investment decisions or to assess the financial health of an organization.
3. **Marketing Analysis:** Quantity analysis is used to evaluate the quantity of demand for a particular product or service. This information is then used to make decisions about pricing, product development, and marketing strategies.
4. **Operations Management:** Quantity analysis is used to evaluate the quantity of inputs and outputs in a production process. This information is then used to optimize production processes and increase efficiency.
5. **Environmental Analysis:** Quantity analysis is used to evaluate the quantity of various environmental factors, such as air pollution or greenhouse gas emissions. This information is then used to make informed decisions about environmental policies and practices.

Advantages of Quantity Analysis

There are several advantages of using quantity analysis in various fields, including:

1. **Objectivity:** Quantity analysis involves the use of numerical data and statistical methods, which provide a more objective and precise evaluation of the quantity of a particular item.
2. **Efficiency:** Quantity analysis can be conducted relatively quickly and efficiently, especially with the use of modern technology and software.
3. **Informed Decision-Making:** The insights obtained from quantity analysis can be used to make informed decisions about various issues, such as pricing, production processes, or environmental policies.

4. **Comparison:** Quantity analysis can be used to compare the quantity of a particular item over time or between different groups or populations, providing valuable insights into trends and patterns.

Limitations of Quantity Analysis

Despite its advantages, quantity analysis also has some limitations, including:

1. **Limited Scope:** Quantity analysis only provides information about the quantity of a particular item and may not take into account other factors, such as quality or context.
2. **Data Availability:** The accuracy and reliability of quantity analysis depend on the availability and quality of data. If data is incomplete or biased, it can lead to inaccurate or misleading results.
3. **Simplification:** Quantity analysis involves the simplification of complex phenomena into numerical data, which may not fully capture the complexity of the issue being analyzed.
4. **Subjectivity:** Despite the use of numerical data and statistical methods, there is still some subjectivity involved in quantity analysis, such as in the selection of variables or the interpretation of results.

CONCLUSION

Quantity analysis is a crucial approach used to measure and evaluate the quantity of items, goods, or services in various fields, such as economics, finance, and business. It involves the collection, processing, and interpretation of data to provide insights into the quantity of a particular item, which can then be used to make informed decisions. There are various methods used in quantity analysis, including quantitative and qualitative methods. Quantitative methods involve the use of numerical data and statistical techniques, while qualitative methods involve the use of non-numerical data and descriptive techniques. Quantity analysis has numerous applications in various fields, including economic analysis, financial analysis, marketing analysis, operations management, and environmental analysis. It has several advantages, such as objectivity, efficiency, informed decision-making, and comparison. However, it also has some limitations, such as limited scope, data availability, simplification, and subjectivity.

REFERENCES

[1] R. J. Gettens, “Maya Blue: An Unsolved Problem in Ancient Pigments,” *Am. Antiq.*, 1962, doi: 10.2307/277679.

[2] B. K. Aichernig *et al.*, “Learning and statistical model checking of system response times,” *Softw. Qual. J.*, 2019, doi: 10.1007/s11219-018-9432-8.

[3] H. Wang, D. Zhong, T. Zhao, and F. Ren, “Integrating Model Checking with SysML in Complex System Safety Analysis,” *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2892745.

[4] M. Deng, H. Cao, W. Zhu, H. Wu, and Y. Zhou, “Benchmark tests for the model-checking-based ids algorithms,” *IEEE Access*, 2019, doi: 10.1109/ACCESS.2019.2939011.

[5] A. Faanhof, J. R. W. Woittiez, and H. A. Das, “Errors in instrumental neutron activation analysis based on short-lived radionuclides,” *J. Radioanal. Chem.*, 1981, doi: 10.1007/BF02517349.

[6] A. L. Treaster, P. P. Jacobs, and G. B. Gurney, “Correcting for the sidewall boundary layer in subsonic two-dimensional airfoil/hydrofoil testing.,” 1981.

[7] “The appliction of elelction quantitives analysis,” *J. Soc. Chem. Ind.*, 1889, doi: 10.1002/jctb.5000080406.

[8] K. Fujie and K. Yoshinoya, “Cytological studies on the cause of pancreatic secretion,” *Arch. Histol. Jpn.*, 1950, doi: 10.1679/aohc1950.1.397.

[9] S. F. Nottingham, D. D. Wilson, R. F. Severson, and S. J. Kays, “Feeding and oviposition preferences of the Sweet Potato Weevil, Cylas formicarius elegantulus, on the outer periderm and exposed inner core of storage roots of selected Sweet Potato cultivars,” *Entomol. Exp. Appl.*, 1987, doi: 10.1111/j.1570-7458.1987.tb01094.x.

[10] T. W. Purcell, “Local Institutions in Grassroots Development: The Rotating Savings and Credit Association,” *Soc. Econ. Stud.*, 2000.

CHAPTER 17

THE IMPLICIT FUNCTION ON INVARIANTS AND TEMPORAL LOGIC TECHNIQUE

Mr. Puneet Kalia, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-puneet.kalia@jnujaipur.ac.in

ABSTRACT:

Invariants and temporal logic are fundamental concepts in computer science and formal methods that play a vital role in verifying the correctness of computer systems. Invariants are properties that remain constant throughout the execution of a program, while temporal logic is a formal language used to specify and reason about the behavior of a program over time. Invariants are used to ensure that a program's behavior remains consistent throughout its execution. They can be used to detect bugs, diagnose errors, and prove the correctness of a program. Invariants are often expressed in terms of mathematical formulas or logical expressions, and they are typically verified using theorem proving techniques or model checking tools.

KEYWORDS:

Computer System, Diagnosis Error, Invariant logic, Temporal Logic, Logical Expression.

INTRODUCTION

In computer science and formal methods, invariants and temporal logic are two important concepts that are used to describe and verify the behavior of complex systems. Invariants are properties of a system that remain unchanged throughout its execution, while temporal logic is a formal language that allows us to express temporal relationships between events and states in a system [1], [2].

In this article, we will explore the concepts of invariants and temporal logic in more detail, including their definitions, types, and applications in computer science.

Invariants

An invariant is a property of a system that remains unchanged throughout its execution. In other words, an invariant is a statement that is always true, regardless of the state of the system. Invariants can be used to describe the behavior of a system and to check whether the system meets certain requirements.

Invariants can be classified into several types, including:

1. **Safety Invariants:** Safety invariants specify conditions that must be true at all times during the execution of a system. These invariants are concerned with preventing errors or unwanted behavior in a system. For example, a safety invariant for a banking system might be that the total balance of all accounts must always be greater than zero.
2. **Liveness Invariants:** Liveness invariants specify conditions that must eventually become true during the execution of a system. These invariants are concerned with ensuring

progress in a system. For example, a liveness invariant for a web server might be that it must always respond to client requests within a certain amount of time.

3. **Invariants over Time:** Invariants over time specify conditions that must be true at all points in time during the execution of a system. These invariants are concerned with the evolution of a system over time. For example, an invariant over time for a traffic control system might be that there can never be more than a certain number of cars in a particular lane at any point in time.

Invariants can be used to verify the correctness of a system by checking whether the system satisfies the desired properties. Invariant checking can be performed using various techniques, including model checking, static analysis, and dynamic analysis [3].

Temporal Logic

Temporal logic is a formal language that allows us to express temporal relationships between events and states in a system. Temporal logic is used to specify properties of a system that depend on the ordering of events and the evolution of the system over time[4]–[6].

There are several types of temporal logic, including:

1. **Linear Temporal Logic (LTL):** LTL is a temporal logic that is used to describe the temporal ordering of events in a system. LTL is based on the concept of linear time, in which events occur in a single sequence. LTL is used to specify temporal properties of a system, such as "eventually," "always," and "until."

2. **Computation Tree Logic (CTL):** CTL is a temporal logic that is used to describe the branching structure of events in a system. CTL is based on the concept of a computation tree, in which events can lead to multiple possible outcomes. CTL is used to specify temporal properties of a system, such as "there exists a path" and "for all paths."

3. **Real-Time Temporal Logic (RTL):** RTL is a temporal logic that is used to describe the timing properties of a system. RTL is based on the concept of real-time, in which events occur at specific points in time. RTL is used to specify temporal properties of a system, such as "before," "after," and "within a certain amount of time."

Temporal logic is used to describe and verify the behavior of complex systems, such as software and hardware systems. Temporal logic can be used to specify temporal properties of a system, such as correctness, liveness, and safety. Temporal logic can be applied to various techniques, such as model checking theorem proving, and deductive reasoning. The concepts of invariants and temporal logic are widely used in computer science and related fields. Here are some examples of their applications in practice:

1. **Software Verification:** Invariants and temporal logic are used in software verification to check whether a program meets its specification. Invariant checking and model checking are common techniques used in software verification to ensure the correctness and safety of a program.

2. **Hardware Verification:** Invariants and temporal logic are used in hardware verification to check the correctness of a hardware design. Formal verification techniques, such as

model checking and theorem proving, are used to ensure the correctness and safety of a hardware design [7], [8].

3. **Robotics:** Invariants and temporal logic are used in robotics to specify and verify the behavior of robotic systems. Temporal logic is used to specify temporal properties of the system, such as reaching a goal or avoiding obstacles. Invariant checking and model checking are used to ensure the correctness and safety of the robotic system.
4. **Cyber security:** Invariants and temporal logic are used in cybersecurity to detect and prevent attacks. Invariant checking and temporal logic can be used to detect vulnerabilities in a system and to verify that security policies are being enforced.
5. **Control Systems:** Invariants and temporal logic are used in control systems to specify and verify the behavior of the system. Temporal logic is used to specify temporal properties of the system, such as stability and convergence. Invariant checking and model checking are used to ensure the correctness and safety of the control system.

DISCUSSION

In computer science, invariants are conditions that must always be true for a system, algorithm, or data structure. Temporal logic is a formal language used to reason about time-based properties of computer systems. Invariants and temporal logic are both important concepts in computer science, and are often used together to specify and verify the correctness of software systems. In this essay, we will explore invariants, temporal logic, and their relationship to each other. An invariant is a condition that must always hold true for a system, algorithm, or data structure. In other words, an invariant is a property that is guaranteed to be true at any point during the execution of a program. Invariants are useful for ensuring the correctness of a program, as they can be used to verify that the program is behaving as expected.

Invariants can be classified into two main types: loop invariants and class invariants. A loop invariant is a condition that must hold true at the beginning and end of each iteration of a loop. Loop invariants are useful for verifying that a loop terminates correctly and produces the desired result. A class invariant is a condition that must hold true for a class, or an object of that class, at all times. Class invariants are useful for ensuring that the state of an object remains consistent and valid throughout its lifetime.

Invariants can be specified using formal or informal methods. Informal methods include comments in code or documentation, and can be useful for providing a general understanding of the program. Formal methods include mathematical notations, such as logical expressions, and can be used for proving the correctness of a program. Temporal logic is a formal language used to reason about time-based properties of computer systems. Temporal logic is used to specify the behavior of a system over time, and to verify that the system behaves correctly. Temporal logic is based on the idea that a system can be divided into a sequence of states, and that the behavior of the system can be described in terms of transitions between these states.

There are two main types of temporal logic: linear temporal logic (LTL) and branching temporal logic (CTL). LTL is used to reason about sequences of states, while CTL is used to reason about the branching structure of a system. In temporal logic, properties are specified using logical expressions that describe the behavior of the system over time. Temporal logic includes operators such as "always" and "eventually" to specify properties that must always hold or must eventually hold, respectively. In Figure 1 illustrate the generating linear temporal logics.

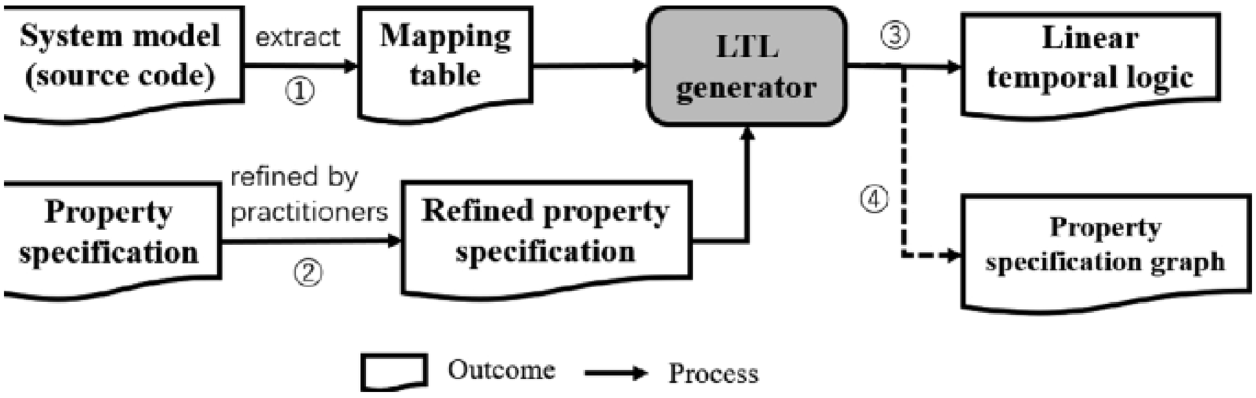


Figure 1: Illustrate the Generating Linear Temporal Logics Based on Property Specification Templates.

Invariants and temporal logic are closely related, as invariants can be expressed using temporal logic. Invariants are properties that must hold at all times during the execution of a program, while temporal logic is used to reason about the behavior of a system over time. By specifying invariants using temporal logic, we can reason about the behavior of a program and verify that the program behaves correctly.

For example, consider a program that implements a stack data structure. An invariant for the stack might be that the stack is always non-empty. This invariant can be expressed using temporal logic as follows: "always, if the stack is not empty, then the top of the stack is defined". This invariant ensures that the stack is always in a valid state, and that the top of the stack is defined whenever the stack is not empty.

Invariants and temporal logic are also useful for detecting and correcting errors in a program. By specifying invariants and verifying them using temporal logic, we can detect errors that violate these invariants. For example, if we specify an invariant for a sorting algorithm that the output is always sorted, we can use temporal logic to verify that the output of the algorithm is indeed sorted. If the output is not sorted, we can use the invariant to help identify

Invariants and temporal logic are also useful for specifying and verifying concurrent systems. Concurrent systems are those in which multiple processes or threads run simultaneously, and can interact with each other. Invariants and temporal logic can be used to ensure that concurrent systems behave correctly, by specifying the properties that must hold for the system as a whole.

For example, consider a concurrent system that consists of two threads that share a data structure. An invariant for this system might be that the data structure is always in a consistent state, and that no thread can access the data structure while it is being modified by another thread. This invariant can be expressed using temporal logic as follows: "always, if a thread is modifying the data structure, then no other thread can access the data structure".

By specifying this invariant and verifying it using temporal logic, we can ensure that the concurrent system behaves correctly, and that the data structure remains in a consistent state. If the invariant is violated, we can use the invariant to help identify the source of the error and correct the program. In Figure 2 show the bounded model for metric temporal logic properties.

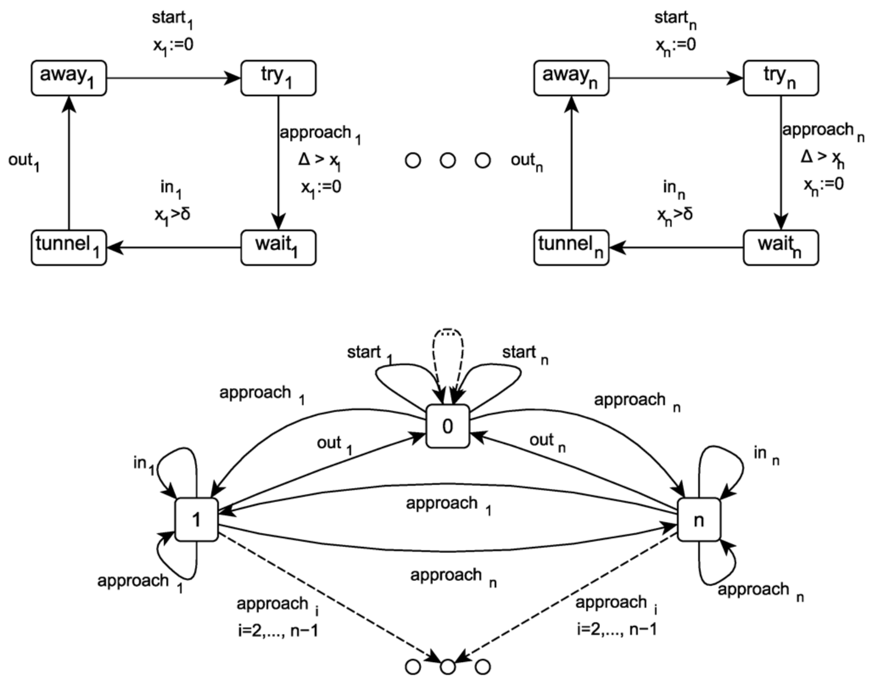


Figure 2: Illustrate the Bounded Model Checking for Metric Temporal Logic Properties of Timed Automata with Digital Clocks.

Invariants and temporal logic are also useful for specifying and verifying real-time systems. Real-time systems are those that must meet strict timing requirements, such as systems used in aerospace, automotive, or medical applications. Invariants and temporal logic can be used to ensure that real-time systems meet their timing requirements, by specifying and verifying the properties that must hold for the system to behave correctly.

For example, consider a real-time system that controls the operation of a medical device. An invariant for this system might be that the system always responds to user input within a specified time interval, and that the system always produces output within a specified time interval. This invariant can be expressed using temporal logic as follows: "always, if a user input is received, then the system must respond within a specified time interval, and if a system output is generated, then it must be produced within a specified time interval".

By specifying this invariant and verifying it using temporal logic, we can ensure that the real-time system behaves correctly, and that it meets its timing requirements. If the invariant is violated, we can use the invariant to help identify the source of the error and correct the program.

Every embedded system has to be created to adhere to particular specifications. These system specs may also be referred to as attributes. System requirements are often described in a language that is familiar to engineers today, such English. Take, for instance, the communication protocol known as SpaceWire, which is being adopted by a number of national space agencies.

Below are two attributes that are taken directly of the specification paper and provide requirements for how the system should operate after a reset:

The ErrorReset state must be reached after a system reset, when a link operation has been stopped for any reason, or if a link initialization error occurs. When the reset signal is asserted, the state machine must transition right away to the ErrorReset state and stay there until the reset signal is de-asserted.

To prevent the ambiguities that are present in natural languages, it is critical to clarify needs accurately. Consider the first SpaceWire protocol characteristic listed above as an example. Keep in mind that the timing of entering the ErrorReset state is not specified. The systems that implement the SpaceWire protocol are synchronous, which means that state machine transitions take place in response to system clock ticks. In light of this, is it necessary to enter the ErrorReset state on the same second the clock ticks once one of the three requirements is met, or on a later tick? It turns out that the paper wants for the system to switch to ErrorReset on the very next tick, although the English language explanation is vague about this.

This chapter will provide methods for accurately and quantitatively defining system attributes. Formal specifications are mathematical descriptions of a system's characteristics. We shall use a particular formalism known as temporal logic. As the name implies, temporal logic uses a precise mathematical language and corresponding rules to express and analyse timing-related system characteristics. While philosophers and logicians have utilised temporal logic since the time of Aristotle, it has only recently been employed as a mathematical language for describing system needs.

An invariant is one of the most prevalent types of system attribute. In terms of a temporal logic quality, it is also one of the most basic types. We shall first define the term "invariant" before expanding it to include more detailed temporal logic requirements. An invariant is a property that applies to a system if it holds true continuously while the system is in use. Think about the traffic light controller model in think about the system that is created when these two state machines are combined asynchronously. The assembled system must, of course, meet the obvious feature that no pedestrians cross while the signal is green (when cars are allowed to move). This condition is system invariant because it must always apply to this system.

It is also preferable to include the invariant characteristics of embedded systems' software and hardware implementations. Several of these qualities outline appropriate linguistic construct use in programming. As an example, the C language invariant "The programme never dereferences a null pointer" describes excellent programming practises. Dereferencing a null pointer often causes a segmentation fault in a C programme, which might result in a system crash.

The thread B that has the lock must not be blocked while attempting to acquire a lock held by A if thread A blocks while attempting to acquire the lock. Each multithreaded programme created from threads A and B must have this characteristic as an invariant. The asset could or might not hold for a certain software. Deadlock might result if it collapses. Moreover, many system invariants put constraints on programme data, as seen in the example below.

We now provide a formal definition of temporal logic and demonstrate how it may be used to define system behaviour using examples. We focus specifically on linear temporal logic (LTL), a subset of temporal logic. Several types of temporal logic exist, some of which are skimmed in sidebars. A property across a single, arbitrary execution of a system may be expressed using LTL.

The following kind of properties, for example, may be expressed in LTL:

a. An event's occurrence and characteristics. For instance, one might state that an event A must occur either infinitely many times or at least once in every trace of the system.
b. The causal connection between occurrences. The rule that states if event A happens in a trace, event B must follow suit is an illustration of this.
c. The sequence of occurrences. A property of this kind may state that each time event A occurs, event B must also occur at the same time.

The preceding intuition on the types of characteristics expressible in linear temporal logic is now formalised. Fixing a certain formal model of computation is beneficial in order to execute this formalisation. The theory of finite-state machines, will be used. It was stated that an execution trace of a finite-state machine is a series of the type q0, q1, q2, q3,..., where qj = (xj, sj, yj), xj is the input valuation, and yj is the output value at response j. Sj is the state in this case. In order to discuss the circumstances at each reaction, such as whether an input or output is present, what the value of an input or output is, or what the state is, we must first be able to define those criteria. Let such a claim about the inputs, outputs, or states be an atomic proposition. It is an antecedent an expression that evaluates to true or false. Atomic propositions that are pertinent for the state machines.

CONCLUSION

Invariants and temporal logic are essential concepts in computer science and formal methods that are used to ensure the correctness of computer systems. Invariants provide a way to verify the consistency of a program's behavior, while temporal logic allows developers to reason about the behavior of a program over time. Together, these concepts provide a powerful set of tools for ensuring the reliability and safety of computer systems.

REFERENCES

[1] N. Roohi, R. Kaur, J. Weimer, O. Sokolsky, and I. Lee, “Parameter invariant monitoring for Signal Temporal Logic,” in *HSCC 2018 - Proceedings of the 21st International Conference on Hybrid Systems: Computation and Control (part of CPS Week)*, 2018. doi: 10.1145/3178126.3178140.

[2] F. Lin, “A formalization of programs in first-order logic with a discrete linear order,” *Artif. Intell.*, 2016, doi: 10.1016/j.artint.2016.01.014.

[3] M. J. Zyphur *et al.*, “From Data to Causes II: Comparing Approaches to Panel Data Analysis,” *Organ. Res. Methods*, 2020, doi: 10.1177/1094428119847280.

[4] T. Dreossi, A. Donzé, and S. A. Seshia, “Compositional Falsification of Cyber-Physical Systems with Machine Learning Components,” *J. Autom. Reason.*, 2019, doi: 10.1007/s10817-018-09509-5.

[5] A. Cailliau and A. Van Lamsweerde, “Runtime monitoring and resolution of probabilistic obstacles to system goals,” *ACM Trans. Auton. Adapt. Syst.*, 2019, doi: 10.1145/3337800.

[6] B. Sheehey, “Algorithmic paranoia: the temporal governmentality of predictive policing,” *Ethics Inf. Technol.*, 2019, doi: 10.1007/s10676-018-9489-x.

[7] S. Haesaert, P. M. J. Van den Hof, and A. Abate, “Data-driven and model-based verification via Bayesian identification and reachability analysis,” *Automatica*, 2017, doi: 10.1016/j.automatica.2017.01.037.

[8] A. U. Shankar, “An Introduction to Assertional Reasoning for Concurrent Systems,” *ACM Comput. Surv.*, 1993, doi: 10.1145/158439.158441.

CHAPTER 18

SIMPLIFIED DETERMINATION ALGORITHM ON EXECUTION TIME ANALYSIS

Mr. Anil Agarwal, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-anil.agarwal@jnujaipur.ac.in

ABSTRACT:

Execution time analysis refers to the process of measuring and evaluating the amount of time required for a program or process to run to completion. It is an important aspect of software development and performance optimization, as it provides insight into how efficient a program is and helps identify areas for improvement. In execution time analysis, developers use various tools and techniques to measure the time it takes for a program to complete specific tasks or functions. They may use profiling tools to identify bottlenecks or areas where the program is spending a lot of time, or they may use benchmarks to compare the performance of different implementations.

KEYWORDS:

Analysis, Benchmark, Developers, Execution Time, Performance Optimization.

INTRODUCTION

The phrase is either true or untrue at a response qi in each situation. If qi = (x, b, y) for any values of x and y, then the machine is in state b at the beginning of the reaction, and the proposition b is true at reaction qi. Hence, it speaks about the present condition rather than the one to come. A predicate that combines atomic assertions using the logical connectives conjunction (logical AND, denoted), disjunction (logical OR, denoted), negation (logical NOT, indicated), and implies (logical implication, denoted =) is known as a propositional logic formula or (more simply) proposition. Any of the atomic propositions listed above, as well as formulations that combine atomic propositions with logical connectives, may be used to describe the state machines [1].

Keep in mind that if p1 and p2 are propositions, the statement p1 = p2 is only true if p2 = p1. In other words, it is as legitimate to show that p2 = p1 is true if we want to prove that p1 = p2. The latter phrase is known as the contrapositive of the former in logic. Furthermore keep in mind that if p1 is false, p1 = p2 is true. Considering the contrapositive makes this clear. If p1 is true, then the statement "p2 Equals p1" is true regardless of p2. Contrary to the aforementioned statements, an LTL formula applies to a complete trace (q0, q1, q2,...) rather than to a single response (qi). The simplest LTL formulations are identical to the statements above in appearance, but they apply to the whole trace rather than just a particular trace element[2], [3].

If p is a proposition, then it follows that p must be true for q0 in order for the LTL formula = p to hold for the trace q0, q1, q2,... Although if the statement only holds for the first element of the trace, it may seem strange to assert that the formula holds for the whole trace. Yet, we shall show that LTL offers means to reason about the entire trace. It is necessary to provide one trace for which an LTL formula is untrue in order to show that it is false for an FSM. A counterexample is a trace like that. To prove that an LTL formula holds true for an FSM, one must show that it holds true for all traces, which is frequently much more difficult (though not when the LTL formula is a straightforward propositional logic formula, in which case we only need to take into account the first element of the trace).

The LTL formula y is incorrect. A trace where x is not present in the first response serves as a counterexample in both scenarios. LTL formulations may additionally include one or more unique temporal operators in addition to propositions [4], [5]. Since they allow for reasoning about complete traces rather than merely making claims about the initial element of a trace, they make LTL considerably more interesting. We now discuss the four primary temporal operators.

The property G, or "globally G," holds for a trace if G holds for each of the trace's suffixes. (A tail of a trace known as a suffix includes all reactions that follow the initial reaction.) G holds for the trace in mathematical notation if and only if, for every j 0, formula G holds in the suffix qj, qj+1, qj+2, etc. Instance 13.6 G(x = y) holds for the machine [6], [7].(b) because it holds for all traces of the machine. G(x y) is false for every trace where x is missing in any reaction, hence it does not hold for the machine. A trace like that serves as a counterexample.

G simply denotes that if is a propositional logic formula, holds in every response. Yet when we combine the G operator with additional temporal logic, we will observe LINEAR TEMPORAL LOGIC operators, we may make considerably more interesting claims about traces and state machines. If holds for one of the trace's suffixes, then the attribute F (which may be interpreted as "eventually" or "finally") holds for the trace. Technically, formula F holds in the suffix qj, qj+1, qj+2,... if and only if, for any j 0, formula F holds for the trace.

Consider that when reading an LTL formula, parenthesis might be crucial. For instance, if Fb holds for all traces (since the beginning state is b), (Gx) = (Fb) is trivially true. Keep in mind that F is true if and only if G. That is, claiming that something is ultimately untrue is equivalent to claiming that something is sometimes false [8], [9]. If and only if holds for the trace q1, q2, q3, etc., does the property X (which may be interpreted as "next state X") hold for a trace q0, q1, q2, etc.

For this, he received the 1996 ACM Turing Award, the field of computer science's highest prize. Since the publication of his key study, the use of temporal logic to define the attributes of many systems, such as computer hardware, software, and cyber-physical systems, has grown significantly.

DISCUSSION

While LTL has been the emphasis of this chapter, there are other options. Individual traces of an FSM may be described by LTL formulas, and in this chapter, we declare that an LTL formula is valid for an FSM if it is valid for all feasible traces of the FSM. Computation tree logic (CTL), a more broad kind of logic, explicitly offers quantifiers over potential FSM traces (Emerson and Clarke, 1980; Ben-Ari et al., 1981). Instead of requiring that a property hold for all traces, we may, for instance, create a CTL expression that holds for an FSM if there is any trace that meets the requirement. When an FSM response has a nondeterministic choice, CTL is known as a branching-time logic since it will concurrently analyse all possibilities.

LTL, in contrast, is referred to as a linear-time logic since it only takes into account one trace at a time. As LTL does not support quantifiers like "for all traces," our norm of claiming that an LTL formula holds for an FSM if it holds for all traces cannot be represented in LTL. To use this convention, we must depart from logic. This convention may be expressed directly in the logic using CTL.

There are other more temporal logic variations that are useful. For instance, realtime temporal logics (such timed computation tree logic or TCTL) are used to reason about real-time systems where time is not measured in discrete steps but is instead continuous (Alur et al., 1991; Alur and Henzinger, 1993). Similar to how signal temporal logic has been successful in explaining the real-time behaviour of hybrid systems, probabilistic temporal logics are beneficial for thinking about probabilistic models like Markov chains or Markov decision processes (see, for instance, Hansson and Jonsson (1994)). (Maler and Nickovic, 2004).

Since it does not hold for any suffix that starts in state a, Xa) does not hold for the state machine. G(b = Xa) holds for the state machine. A trace has the feature 1U2 (which may be translated as "1 till 2") if 2 is true for some of its suffixes and 1 is true until 2 is true. Technically, if and only if j 0 exists, 2 holds in the suffixes qj, qj+1, qj+2,... and 1 holds in the suffixes qi, qi+1, qi+2,... for every I s.t. 0 I j, 1U2 holds for the trace. 1 may or might not be true for qj, qj+1, qj+2, etc. Let p and q represent the robot encountering an obstacle and the robot being at least 5 cm away from the obstruction, respectively. In LTL, this characteristic may therefore be stated as G (p = Fq).

Think about SpaceWire property 13.12, which states that "Once the reset signal is asserted, the state machine should go directly to the ErrorReset state and stay there until the reset signal is de-asserted." Let q be true while the FSM is in the ErrorReset state and p be true when the reset signal is asserted. The English property mentioned above is then codified in LTL as G (p = X(q U p)). In Figure 1 illustrate the analysis and measurement of time.

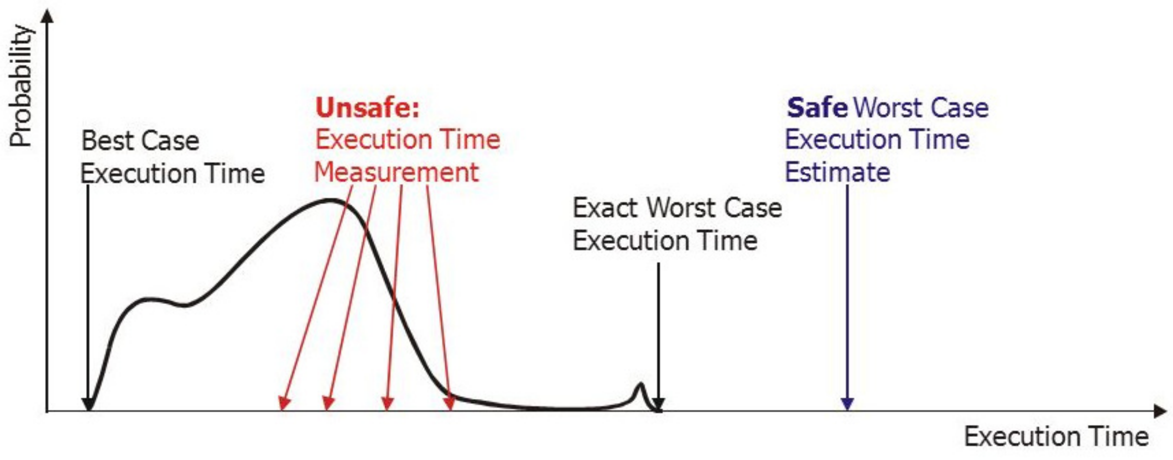


Figure 1: Illustrate the Analysis vs. Measurement Time.

We have taken "immediately" in the formalisation above to indicate that the state switches to ErrorReset in the very next time step. Moreover, any execution where the reset signal is asserted and not afterwards de-asserted will result in the above LTL formula failing to hold. The English language statement does not make it apparent that the standard intended for the reset signal to finally be de-asserted.

The following LTL formulae express qualities that are often helpful.

a. Infinitely numerous instances: This characteristic is of the type G Fp, which denotes that p will always be true at some point. In other words, this proves that p is true indefinitely.
b. Steady-state property: This property has the form F Gp and may be interpreted as "p holds at all times from some point in the future." This is an example of a steady-state property, showing that the system eventually achieves a steady state where p is always true.
c. Request-response property: According to the formula G (p = Fq), a request p will ultimately result in a response q.

In the design of embedded systems, dependability and accuracy are crucial considerations. In turn, formal specifications are essential to achieve these objectives. Temporal logic, one of the primary methods for creating formal specifications, has been covered in this chapter. Techniques for accurately describing a system's attributes that must endure across time have been offered in this chapter. It has concentrated particularly on linear temporal logic, which may describe numerous system safety and liveness features.

Safety or liveliness attributes are examples of system properties. A safety property is technically defined as one that states that "nothing terrible occurs" during execution. Similarly, a liveness attribute guarantees that "positive things will happen" while an application is being executed. A system execution does not fulfil a property p if and only if a finite-length prefix of the execution does not exist that can be extended to an endless execution satisfying p. If any finite-length execution trace can be extended to an infinite execution that satisfies p, then we say that p is a liveness condition.

Contrarily, liveness attributes outline a system's performance or development needs. A property of the type F is a liveness property for a state machine. There is no finite execution that can prove that this property is not met. An even more detailed illustration of a liveness characteristic is as follows: The associated interrupt service routine (ISR) is ultimately run whenever an interrupt is asserted. In temporal logic, this property may be represented as G(p1 = Fp2) if p1 is the property that an interrupt is asserted and p2 is the property that the interrupt service routine is run. Keep in mind that qualities like safety and liveness may both be system invariants. An example of an invariant is the liveness condition on interruptions mentioned earlier; p1 = Fp2 must hold in all states.

There are two types of liveliness properties: bound and unbound. A bounded liveness property which is a safety feature sets a temporal limit on a desired event occurring. In the aforementioned example, the property is a limited liveness property if the ISR must be performed within 100 clock cycles of the interrupt being asserted; otherwise, if there is no such time constraint on the occurrence of the ISR, it is an unbounded liveness property. The X operator may be used by LTL to represent certain constrained liveness features, but it does not provide a way to directly measure time.

The sequence w satisfies the following temporal logic formula for each conceivable behaviour of the composition, however none of the other sequences that are not behaviours of the composition do so: Justify your answer. If you conclude it is incorrect, then offer a temporal logic formula \sfor which the statement is true. In this issue, tasks to be carried out by a robot are specified using linear temporal logic. Assume the robot has to go to a series of n sites, such as l1, l2,..., ln.

Let pi represent an atomic formula that is valid only if the robot travels to the specified point, li.

Provide LTL formulae for the following assignments:

(a) At some point, the robot must stop by at least one of the n sites.
(b) In any sequence, the robot must ultimately visit all n places.
(c) In the sequence l1, l2,..., ln, the robot must finally visit all n places.

M may operate in two different modes: inactive, where the main programme runs, and active, where the interrupt service routine (ISR) runs. A shared variable called timerCount is read and updated by the main programme and ISR.

Answer the following: (a) Using appropriate atomic propositions, specify the following characteristic in linear temporal logic: Program location C is finally reached by the main programme.

Explain your response by building the product FSM. What circumstances would M meet the property if it doesn't now? Suppose that M's environment has perpetual access to the interrupt Clearly state your presumptions. Have a look at the software fragment, which offers methods for asynchronous thread communication via message sending. In Figure 2 illustrate the techniques for measuring execution time.

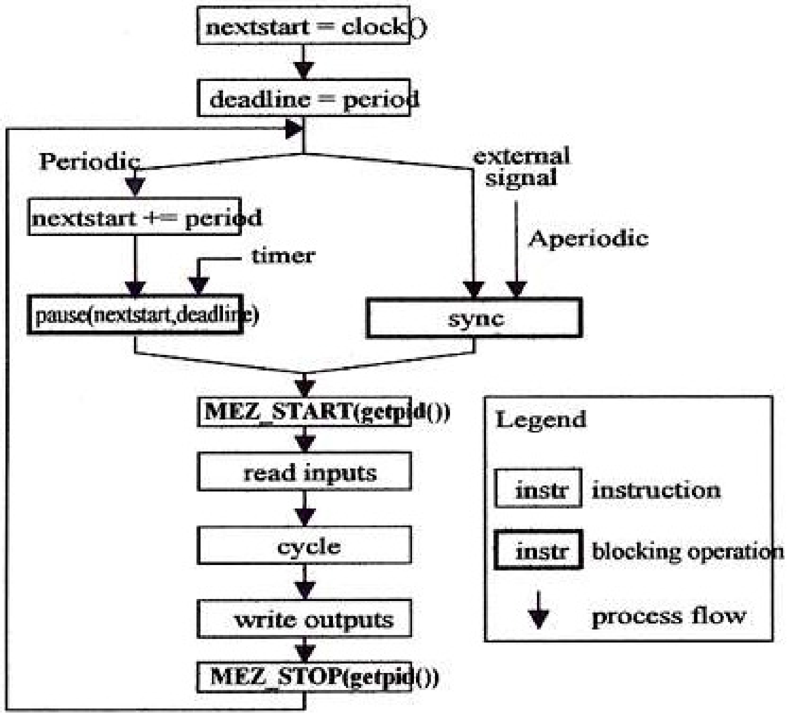


Figure 2: Illustrate the Techniques for measuring execution time and real time performance.

Please respond to the following inquiries about this code. Assuming there is just one CPU powering the code (not a multicore machine). You may alternatively presume that the static variables shown are solely accessible by the code displayed.

(a) Let s to be an atomic proposition stating that send executes line 24 and releases the mutex. Suppose g is an atomic proposition that states get releases the mutex (i.e. executes line 38). Provide an LTL formula that states that during a programme execution, g cannot come before s. Does this formula hold for any software using these processes for the first time?

(b) Assume that a programme that makes use of the send and get commands in Figure 11.6 is terminated at any time and then resumed from the beginning. A call might get to return in the new execution even before a call to transmit has been placed. Explain a scenario where this may occur.

(c) Assume once again that a programme that makes use of the aforementioned send and get operations is terminated at any time and then resumed from the beginning. Is a stalemate feasible in the new execution, where neither 374 Introduction to Embedded Systems, Lee & Seshia

Neither a call to receive nor a call to transmit may return Invariants and Temporal Logic. If yes, explain how this may happen and provide a solution. If not, provide a defence.

CONCLUSION

The operating system activities' worst-case execution times (WCET) is a crucial step in developing operating systems for demanding real-time applications. This has traditionally been accomplished by thoroughly testing the operating system and paying close attention to the testing circumstances to accurately simulate the worst-case execution situation an alternate method of predicting off-line the WCET of the system calls of a real-time kernel, the RTEMS kernel, using static analysis. We provide qualitative and quantitative findings from the study of RTEMS and make some inferences about the applicability of static analysis to operating system code.

REFERENCES

[1] J. Kim, M. Oh, and J. Kim, "Effect of analysis of algorithm execution time and adopting unplugged method on third grade elementary students' computational thinking ability," *Asia Life Sci.*, 2020, doi: 10.21742/ijcwpm.2019.3.1.02.

[2] S. P. Chinchali, S. C. Livingston, M. Chen, and M. Pavone, "Multi-objective optimal control for proactive decision making with temporal logic models," *Int. J. Rob. Res.*, 2019, doi: 10.1177/0278364919868290.

[3] Z. Xu and U. Topcu, "Transfer of temporal logic formulas in reinforcement learning," in *IJCAI International Joint Conference on Artificial Intelligence*, 2019. doi: 10.24963/ijcai.2019/557.

[4] R. Kirner, J. Knoop, A. Prantl, M. Schordan, and A. Kadlec, "Beyond loop bounds: Comparing annotation languages for worst-case execution time analysis," *Softw. Syst. Model.*, 2011, doi: 10.1007/s10270-010-0161-0.

[5] K. Nürnberger, M. Hochstrasser, and F. Holzapfel, "Execution time analysis and

optimisation techniques in the model-based development of a flight control software," *IET Cyber-Physical Syst. Theory Appl.*, 2017, doi: 10.1049/iet-cps.2016.0046.

[6] J. Engblom, A. Ermedahl, M. Sjödin, J. Gustafsson, and H. Hansson, "Worst-case execution-time analysis for embedded real-time systems," *Int. J. Softw. Tools Technol. Transf.*, 2003, doi: 10.1007/s100090100054.

[7] S. Hahn, J. Reineke, and R. Wilhelm, "Towards Compositionality in Execution Time Analysis: Definition and Challenges," *ACM SIGBED Rev.*, 2015.

[8] P. Černý, T. A. Henzinger, L. Kovács, A. Radhakrishna, and J. Zwirchmayr, "Segment abstraction for worst-case execution time analysis," *Lect. Notes Comput. Sci. (including Subser. Lect. Notes Artif. Intell. Lect. Notes Bioinformatics)*, 2015, doi: 10.1007/978-3-662-46669-8_5.

[9] A. Colin and I. Puaut, "Worst case execution time analysis for a processor with branch prediction," *Real-Time Syst.*, 2000, doi: 10.1023/a:1008149332687.

CHAPTER 19

19 OPTIMIZATION TECHNIQUE FOR EQUIVALENCE AND REFINEMENT

Mr. Puneet Kalia, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-puneet.kalia@jnujaipur.ac.in

ABSTRACT:

Equivalence and refinement are two concepts in computer science that are used to describe the relationship between different models or specifications of a system. Equivalence refers to the idea that two models or specifications are functionally equivalent, meaning that they produce the same output for the same input. Refinement, on the other hand, refers to the process of creating a more detailed or precise model or specification from a higher-level one. In software development, equivalence and refinement are important for ensuring that a system is implemented correctly and performs as expected. Equivalence testing is often used to compare different implementations of a system to ensure that they produce the same results, while refinement is used to progressively refine and improve the specification of a system until it is ready for implementation.

KEYWORDS:

Development, Equivalence, Refinement, Testing, Software.

INTRODUCTION

Equivalence and refinement are important concepts in the field of software engineering and formal methods. They are used to verify the correctness of software systems by comparing different versions of the same system or by relating a high-level specification to a low-level implementation. In this article, we will explore the concepts of equivalence and refinement in detail, and discuss their applications in software engineering.

Equivalence

In software engineering, equivalence refers to the idea that two software systems or components have the same behavior or functionality. Equivalence is an important concept because it allows us to compare different versions of the same system or component and verify that they behave in the same way. There are different types of equivalence that are relevant in software engineering. Some of the most common types are:

1. **Input-output equivalence:** Two systems are input-output equivalent if they produce the same outputs for the same inputs.
2. **Trace equivalence:** Two systems are trace equivalent if they produce the same sequence of events in response to the same inputs.
3. **Behavioral equivalence:** Two systems are behavioral equivalent if they exhibit the same observable behavior.

Equivalence can be established using different techniques, such as testing, model checking, or theorem proving. Testing is a common approach to establish input-output equivalence, where the systems are tested with a set of inputs, and the outputs are compared to verify that they are the same. Model checking and theorem proving are more formal approaches that use mathematical models to establish equivalence between systems.

Refinement

Refinement is the process of transforming a high-level specification into a low-level implementation while preserving the behavior of the system. Refinement is an important concept in software engineering because it allows us to design and implement complex systems in a structured and systematic way. Refinement is often used in the development of safety-critical systems, such as those used in aerospace or medical applications.

The refinement process typically involves several steps, starting with a high-level specification of the system and progressively refining it into a low-level implementation. Each step of the refinement process must preserve the behavior of the system, and the resulting implementation must be correct with respect to the original specification.

There are different levels of refinement that can be applied to a system, such as:

1. **Functional refinement:** Refinement at this level involves transforming a high-level specification into a detailed functional design.
2. **Architectural refinement:** Refinement at this level involves refining the functional design into an architectural design that specifies the system components and their interactions.
3. **Implementation Refinement:** Refinement at this level involves transforming the architectural design into a low-level implementation that can be executed on a target platform.

The refinement process can be supported by different techniques, such as formal methods or model-based engineering. Formal methods use mathematical models and proof techniques to verify the correctness of the system at each step of the refinement process. Model-based engineering, on the other hand, uses models to describe the system at different levels of abstraction and to generate the implementation code [1], [2].

Equivalence and Refinement in Software Engineering

Equivalence and refinement are closely related concepts in software engineering, and they are often used together to verify the correctness of software systems. Equivalence is used to compare different versions of the same system or to relate a high-level specification to a low-level implementation. Refinement, on the other hand, is used to transform a high-level specification into a low-level implementation while preserving the behavior of the system.

Equivalence can be used to verify the correctness of the refinement process. If two systems are input-output equivalent, for example, we can conclude that the refinement process has preserved the behavior of the system. Similarly, if two systems are trace equivalent or behavioral equivalent, we can conclude that the refinement process has preserved the observable behavior of the system [3], [4].

Equivalence can also be used to compare different implementations of the same system. For example, if we have two implementations of the same system, we can use equivalence to verify that they have the same behavior. This can be useful in software maintenance, where we may need to replace an existing implementation with a new one.

Refinement, on the other hand, can be used to ensure that the implementation of a system satisfies its specification. If we have a high-level specification of a system, we can use refinement to systematically transform it into a low-level implementation that is correct with respect to the original specification.

Refinement can also be used to generate test cases for the system. If we have a high-level specification and a low-level implementation of the system, we can use refinement to generate a set of test cases that cover all the possible behaviors of the system. This can be useful in testing and validation of the system [5].

DISCUSSION

Equivalence and refinement are also used in the development of safety-critical systems. Safety-critical systems are those whose failure could result in loss of life, injury, or damage to property. In such systems, it is crucial to ensure that the system behaves correctly and that its behavior can be verified and validated. Equivalence is used to compare the safety-critical system with its safety specification, which specifies the safety properties that the system must satisfy. By establishing equivalence between the system and the safety specification, we can ensure that the safety properties are satisfied by the system.

Refinement is used to ensure that the safety-critical system satisfies its safety specification. The safety specification is usually a high-level specification that specifies the safety properties of the system. Refinement is used to systematically transform the safety specification into a low-level implementation that satisfies the safety properties. Equivalence and refinement can also be used in the verification of concurrent and distributed systems. Concurrent and distributed systems are those that have multiple threads of execution or that are distributed across multiple machines. Verifying the correctness of such systems can be challenging due to their complexity and non-deterministic behavior[6]–[8].

Equivalence is used to compare different implementations of the concurrent or distributed system. For example, we can use input-output equivalence to compare the behavior of two different implementations of the same distributed system. By establishing equivalence between the two implementations, we can ensure that they have the same behavior.

Refinement is used to ensure that the concurrent or distributed system satisfies its specification. The specification of a concurrent or distributed system typically includes the behavior of the system under different execution scenarios. Refinement is used to transform the specification into a low-level implementation that can be executed on the target platform.

Equivalence and refinement are important concepts in software engineering and formal methods. Equivalence allows us to compare different versions of the same system or to relate a high-level specification to a low-level implementation. Refinement allows us to systematically transform a high-level specification into a low-level implementation while preserving the behavior of the system.

Equivalence and refinement are often used together to verify the correctness of software systems. Equivalence can be used to compare different implementations of the same system or to verify that the refinement process has preserved the behavior of the system. Refinement can be used to ensure that the implementation of a system satisfies its specification or to generate test cases for the system.

Equivalence and refinement are also important in the development of safety-critical systems and the verification of concurrent and distributed systems. In all these applications, equivalence and refinement are used to ensure the correctness of the system and to minimize the risk of failure. In Figure 1 illustrate the robust equivalence checking.

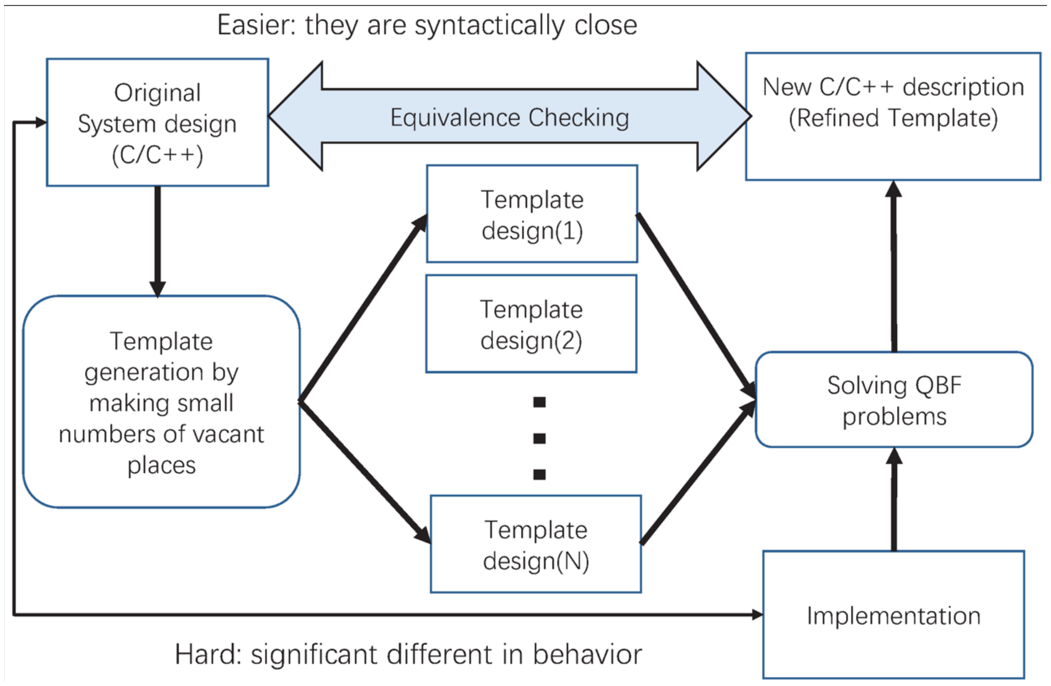

Easier: they are syntactically close
Original System design (C/C++)
Equivalence Checking
New C/C++ description (Refined Template)
Template design(1)
Template design(2)
Template generation by making small numbers of vacant places
Solving QBF problems
Template design(N)
Implementation
Hard: significant different in behavior

Figure 1: Illustrate the Approach to Robust Equivalence Checking

Equivalence and refinement are two important concepts in formal methods, which are used to reason about the correctness of software and hardware systems. In this article, we will explain what these concepts are, and how they are used in practice.

Equivalence

Equivalence is the concept of two systems having the same behavior, despite being represented differently. In formal methods, equivalence is typically used to compare two models or two versions of a system to determine whether they have the same behavior. There are several types of equivalence that are commonly used, including:

1. **Structural equivalence:** Two systems are structurally equivalent if they have the same structure, but may differ in the values of their variables or inputs.

2. **Strong equivalence:** Two systems are strongly equivalent if they have the same structure and the same values of their variables and inputs.
3. **Trace equivalence:** Two systems are trace equivalent if they produce the same set of traces, which are sequences of events that can occur during their execution.
4. **Bisimulation equivalence:** Two systems are bisimulation equivalent if they have the same behavior, and one system can simulate the behavior of the other, and vice versa.

Equivalence is important in formal methods because it allows us to reason about the correctness of a system by comparing it to a model of a correct system. If the two systems are equivalent, we can be sure that the original system is correct.

Refinement

Refinement is the concept of transforming a high-level specification or design into a low-level implementation that satisfies the specification. In other words, refinement is the process of making a system more detailed and more concrete. Refinement is typically used in software engineering and hardware design to translate a high-level design or specification into a low-level implementation.

There are several types of refinement that are commonly used, including:

1. **Data refinement:** Data refinement is the process of adding more detail to the data structures in a system.
2. **Control refinement:** Control refinement is the process of adding more detail to the control flow in a system.
3. **Operation refinement:** Operation refinement is the process of adding more detail to the operations or functions in a system.

Refinement is important in formal methods because it allows us to gradually build up a system from a high-level specification to a low-level implementation, while ensuring that the implementation satisfies the specification. By breaking down the implementation into smaller steps, we can ensure that each step is correct, and that the overall implementation is correct.

Equivalence and Refinement in Practice

Equivalence and refinement are used in practice in several ways. Here are some examples:

1. **Software Verification:** Equivalence and refinement are used in software verification to ensure that a program meets its specification. Formal methods, such as model checking and theorem proving, are used to check the equivalence between the program and the specification, and to refine the program to satisfy the specification.
2. **Hardware Verification:** Equivalence and refinement are used in hardware verification to ensure that a hardware design meets its specification. Formal verification techniques, such as model checking and equivalence checking, are used to check the equivalence between the design and the specification, and to refine the design to satisfy the specification.

3. **Compiler Optimization:** Equivalence and refinement are used in compiler optimization to ensure that the optimized code has the same behavior as the original code. Formal verification techniques, such as equivalence checking, are used to check the equivalence between the optimized code and the original code.
4. **Cryptography:** Equivalence and refinement are used in cryptography to ensure the correctness and security of cryptographic protocols. Formal methods, such as protocol analysis and equivalence checking, are used to ensure that the protocol meets its security requirements.

Equivalence and refinement are important concepts in formal methods, which are used to ensure the correctness of software and hardware systems. Equivalence is the concept of two systems having the same behavior, despite being represented differently, while refinement is the concept of transforming a high-level specification or design into a low-level implementation that satisfies the specification. These concepts are used in a variety of applications, such as software verification, hardware verification, compiler optimization, and cryptography. In Figure 2 illustrate the equivalence checking.

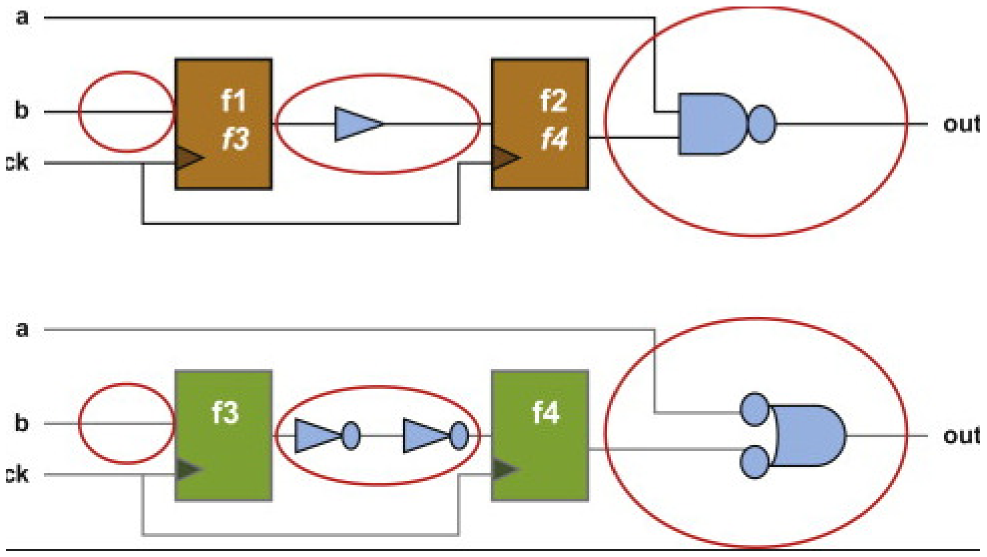


Figure 2: Illustrate the equivalence checking.

Equivalence and refinement are closely related concepts, and are often used together in practice. For example, in software verification, we might use refinement to gradually build up a program from a high-level specification to a low-level implementation, while using equivalence to ensure that each step in the refinement process produces a program that is equivalent to the specification.

In hardware design, we might use equivalence to compare a design to its specification, and use refinement to gradually add more detail to the design until it satisfies the specification. In both cases, equivalence and refinement are used to ensure that the final implementation meets its specification.

One of the key benefits of using equivalence and refinement in formal methods is that it allows us to reason about the correctness of a system at different levels of abstraction. By breaking down a system into smaller components, we can reason about each component in isolation, and then use equivalence and refinement to combine the components into a larger system that satisfies its specification.

Another benefit of using equivalence and refinement in formal methods is that it allows us to catch errors early in the development process. By using equivalence to compare a system to its specification, we can catch errors before they are introduced into the implementation. By using refinement to gradually build up a system from a high-level specification to a low-level implementation, we can catch errors at each step of the refinement process, before they become more difficult to fix.

In conclusion, equivalence and refinement are important concepts in formal methods that are used to ensure the correctness of software and hardware systems. These concepts are closely related, and are often used together in practice to catch errors early in the development process and to reason about the correctness of a system at different levels of abstraction.

The methods for clearly describing the attributes that a system must possess in order to operate correctly and securely were covered in the previous chapter. Linear temporal logic was used to define these qualities because it can succinctly convey the criteria that the trace of a finite-state machine needs to fulfil. An alternate method of expressing requirements is to provide a model or specification that illustrates the intended system behaviour. The specification is often extremely abstract and may display more behaviours than a practical system implementation. Yet, a specification must clearly include undesirable or hazardous behaviours in order to be helpful.

"The lights shall always be lit in the sequence green, yellow, and red," is a straightforward specification for a traffic light. Never, for instance, should it go straight from yellow to green or from green to red. This condition may be expressed as an abstract model or as a formula for temporal logic. This chapter discusses the usage of abstract models as specifications, as well as how these models connect to system implementations and formulae for temporal logic. In this section, we'll explain how to prove that the traffic light model. Also, although not all traces of the specification meet the temporal logic formula, all traces of the model. As a result, these two requirements are different.

This chapter discusses model comparison and how to claim with certainty that one model can be used instead of another. This makes it possible to carry out an engineering design process where we begin with abstract descriptions of desired and undesirable behaviours and gradually improve our models until they are sufficiently comprehensive. Type equivalent and refinement is fully implemented It also indicates when it is safe to substitute one implementation for another, maybe one with a lower implementation cost.

Refinement and Type Equivalence

We start with a straightforward comparison of two models that merely compares the data types used in their communications with the environment. It is specifically intended to prevent data type conflicts from occurring when a model B is used in any situation where a model A is appropriate. B must be able to take any inputs.

CONCLUSION

Equivalence and refinement are two important concepts in computer science that play a crucial role in ensuring the correctness, reliability, and performance of software systems. Equivalence refers to the idea that two models or specifications are functionally equivalent, while refinement involves creating more detailed and precise models from higher-level ones. Equivalence and refinement are used in various areas of software development, such as testing, verification, and implementation. Equivalence testing helps ensure that different implementations of a system produce the same output for the same input, while refinement ensures that a more detailed model is a correct and complete implementation of a higher-level one.

REFERENCES

[1] J. Dyck, H. Giese, and L. Lambers, "Automatic verification of behavior preservation at the transformation level for relational model transformation," *Softw. Syst. Model.*, 2019, doi: 10.1007/s10270-018-00706-9.

[2] C. Baier, M. Sirjani, F. Arbab, and J. Rutten, "Modeling component connectors in Reo by constraint automata," *Sci. Comput. Program.*, 2006, doi: 10.1016/j.scico.2005.10.008.

[3] P. Soffer, "Refinement equivalence in model-based reuse: Overcoming differences in abstraction level," *J. Database Manag.*, 2005, doi: 10.4018/jdm.2005070102.

[4] A. Aguirre, G. Barthe, M. Gaboardi, D. Garg, and P. Y. Strub, "A relational logic for higher-Order programs," *Proc. ACM Program. Lang.*, 2017, doi: 10.1145/3110265.

[5] N. Grimm *et al.*, "A monadic framework for relational verification: Applied to information security, program equivalence, and optimizations," in *CPP 2018 - Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, Co-located with POPL 2018*, 2018. doi: 10.1145/3167090.

[6] S. P. Chinchali, S. C. Livingston, M. Chen, and M. Pavone, "Multi-objective optimal control for proactive decision making with temporal logic models," *Int. J. Rob. Res.*, 2019, doi: 10.1177/0278364919868290.

[7] Z. Xu and U. Topcu, "Transfer of temporal logic formulas in reinforcement learning," in *IJCAI International Joint Conference on Artificial Intelligence*, 2019. doi: 10.24963/ijcai.2019/557.

[8] K. Venkatraman, K. Geetha, and C. Rajan, "A secured software architecture for providing data security in cloud," *J. Adv. Res. Dyn. Control Syst.*, 2019.

CHAPTER 20

ADVANCE OPTIMIZATION IN POWER AND ENERGY ANALYSIS

Mr. Anil Agarwal, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-anil.agarwal@jnujaipur.ac.in

ABSTRACT:

Power and energy analysis is a field of study focused on the measurement and analysis of the energy consumption and efficiency of various systems and devices. This involves the use of various techniques and tools to quantify the amount of power and energy used by different components and systems, and to identify areas where energy consumption can be reduced. One important aspect of power and energy analysis is the study of power quality, which involves the assessment of the various parameters that affect the quality of the power supply, such as voltage, frequency, and harmonic distortion. This is particularly important in the context of renewable energy sources such as wind and solar, which can introduce significant variability in the power supply.

KEYWORDS:

Component System, Energy Analysis, Power Analysis, Voltage, Harmonic Distortion.

INTRODUCTION

The interactions between models known as abstraction and refinement are the main topic of this chapter. The statements "model A is an abstraction of model B" and "model B is a refinement of model A" have the same meaning since these phrases are symmetric. The abstraction A is often simpler, smaller, or easier to comprehend, but the refinement model B contains more information. If attributes that are true of the abstraction are also true of the refinement, then the abstraction is sound with regard to some formal system of properties. A type system, linear temporal logic, or a language of state machines are a few examples of the formal system of properties [1], [2].

A is a reliable abstraction of B if the formal system is LTL and any LTL formula that holds for A also holds for B. This is helpful when proving a formula holds for A is simpler than proving it holds for B, such as when B's state space is much bigger than A's state space.If attributes that are true of the refinement are also true of the abstraction, then the abstraction is complete in terms of some formal system of properties. If the formal system of properties is LTL, for instance, any LTL formula that holds for B also holds for A, then A is a full abstraction of B. Since it is difficult to create a full abstraction that is noticeably simpler or smaller, useful abstractions are often sound but incomplete[3].

Think about a programme B written in an imperative language like C that has several threads, for instance. We may create an abstraction A that disregards variable values and substitutes nondeterministic decisions for all branching and control structures [4]. While the abstraction is obviously less informative than the programme, it may be enough to demonstrate certain of the program's characteristics, such as a mutual exclusion feature.

Every environment that can take any output that A can create may likewise accept any output that B can produce. A can accept from the environment.

Assume an actor model for A and B to give the issue more context, as shown in Figure 14.1. There are three ports on A in that diagram, two of which are input ports denoted by the sets PA = x, w, and one of which is an output port denoted by the sets QA = y. These ports stand for the exchange of information between A and its surroundings. The inputs are of type Vx and Vw, so when the actor reacts, their values will belong to one of the sets Vx or Vw. Four restrictions are imposed by the ports and their kinds if we wish to replace A with B in a certain environment:

1. The first restriction is that B cannot need an input signal that the surroundings do not provide. If the set PB provides the input ports of B, then PB PA ensures that this is the case. A subset of the ports in A are the ports in B. It is safe for A to have more input ports than B because, if B were to take the place of A in a certain environment, it could simply disregard any input signals it did not need.
2. The second restriction is that B must create all output signals that the surrounding environment could need. The restriction QA QB, which states that QA must be the set of output ports of A and QB must be the set of output ports of B, ensures this. A working environment for A does not anticipate such outputs and may thus disregard them, hence it is safe for B to have extra output ports. The two remaining restrictions relate to the different kinds of ports. Let Vp determine the kind of an input port p PA [5], [6]. This indicates that a valid input value v on p meets the condition v Vp. Set the type of an input port p PB to V 0 p.
3. The third restriction states that if the environment offers a value v Vp on an input port p that is acceptable to A, then the value is likewise acceptable to B if p is also an input port of B, i.e., v V 0 p. This restriction may be expressed succinctly as follows: p PB, Vp V 0.Let's say that an output port's type is Vq and its equivalent output port's type is V 0 q.
4. The fourth restriction is that any environment in which A is capable of operating must accept a value produced by B on an output port q if q is also an output port of A. In other words, V 0 q Vq and q QA

When all four conditions are met, B is referred to be a type refinement of A. If B is a type refinement of A, then swapping A out for B in any situation won't lead to type system issues. Of course, additional issues might arise because of the way B is acting. We say that A and B are type equivalent if B is a type refinement of A and A is a type refinement of B. They both feature identical input and output ports, as well as identical port types. Let B stand in for the more in-depth deterministic model [7], [8]. A represent the nondeterministic traffic light model, respectively. Both machines have the same ports and port types, making them type comparable. Hence, substituting A for B or vice versa will not result in type system issues in any scenario. Be aware that it could make sense to remove that port disregards pedestrian input. Let A0 stand in for a version without the input from the pedestrian. B needs a pedestrian signal input, while A0 may be used in settings without one. As a result, it is not always safe to use B in lieu of A0.

DISCUSSION

Looking at the data types of the inputs and outputs alone is often insufficient to replace a machine A with a machine B. A often sets more restrictions than merely data types if A is a specification and B is an implementation. B typically has to conform in some manner to the functionality of A if it is an optimization of A (e.g., a lower cost implementation, a refinement that adds functionality or makes use of new technology).

We examine a more robust equivalence and refinement in this part. Equivalence will specifically imply that the two machines create the same output values when given a certain set of input valuations. These two machines, however, are comparable in a manner that goes well beyond simple type equivalence. From the outside, each of these devices exhibit precisely the same behaviour. The two devices will generate the same output sequence when given the identical input sequence. Think about the port p of a Vp state machine. One value from the set Vp "absent" will be present in this port at each reaction. This sequence may be represented as a function of the form sp: N Vp "missing".

Depending on whether the port is an input or an output, this is the signal that was sent or received there. Remember that assigning such a signal to each port of a state machine results in that machine's behaviour. Remember that the language L(M) of a state machine M is the collection of all of its actions. If two computers speak the same language, they are said to be linguistic equivalents. A series of present and absent values for the two inputs, up and down, along with the appropriate output sequence at the output port, count, is how the garage counter behaves in Example 14.5 Example 3.16 provides a concrete example. The behaviours in Figures 3.4 and 3.8 both exhibit this. Every action in Figure 3.4 is a behaviour in Figure 3.8, and vice versa. These two devices have the same language.

The same input signals may be used by two different behaviours in the case of a nondeterministic machine M. In other words, given an input signal, more than one potential output exists. se382 Introduction to Embedded Systems, Lee & Seshia quence. All potential behaviours are covered by the language L(M). Two nondeterministic computers that speak the same language are linguistically comparable, just as deterministic machines are.

Let's say that L(A) L for two state machines A and B. (B). In other words, B exhibits behaviours that A does not. The term for this is language confinement. It is argued that A is a linguistic improvement over B. Language refinement, like type refinement, makes a claim regarding the acceptability of A as an alternative to B. Every action taken by B will be acceptable in a given context, and every action taken by a will likewise be acceptable in that setting. B may be replaced with A. Figure 1 illustrate the Techno-economic analysis of renewables-driven power.

The FSMs examined in this work have infinitely many possible executions. Assume that only the finite executions are of relevance to us. In order to do this, we provide the idea of an accepting state, denoted by a double outline, as in state b in the example below:

The subset of the language L(M) that emerges from executions that end in an acceptable state is denoted by La(M). In other words, La(M) solely takes into account L(M) behaviours that continue to be in an accepting state while having an endless tail of stuttering responses. As the inputs and outputs will no longer exist after a limited number of reactions, or in LTL, FGp for every port p, all such executions are essentially finite.

The language that an FSM accepts is referred to as La(M). A behaviour in La(M) provides a finite string or finite series of values of type Vp for each port p. The input strings for the aforementioned example are (1), (1, 0, 1), (1, 0, 1, 0, 1), etc., all in La (M). Versions of these include also those in which any two present values may be separated by an arbitrary finite number of missing values. These strings 1, 101, 10101, etc. may be written when there is no ambiguity. The output in the aforementioned example is present a limited number of times in the same responses as the input in all behaviours in La(M).

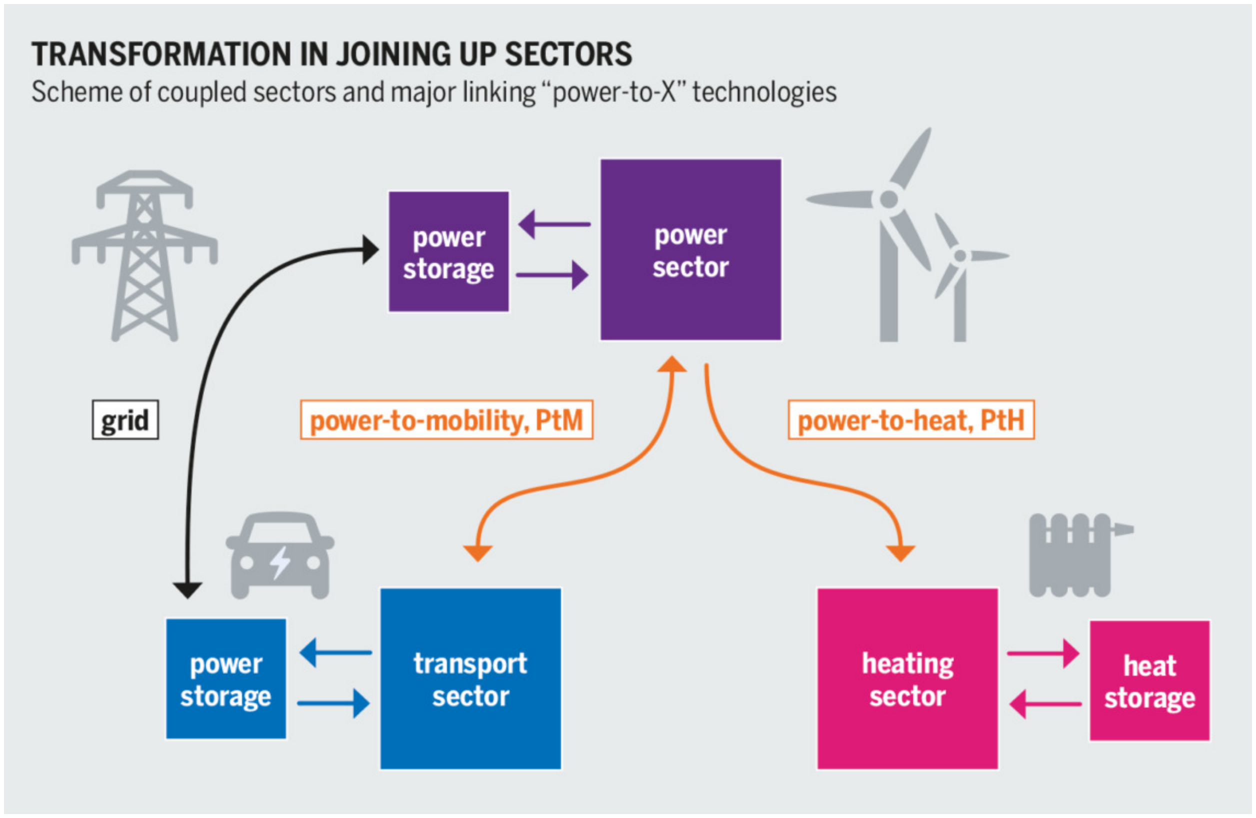


Figure 1: Illustrate the Techno-economic analysis of renewables-driven power.

The state machines described in this article are receptive, which means that each input port p may take any value from its type Vp or be empty at each reaction. Hence, all potential input valuation sequences are included in the language L(M) of the machine described above. Any of these that do not leave the machine in an accepted condition are excluded by La(M). For instance, the infinite sequence (1, 0, 1, 0,) and any input sequence with two 1s in a succession are in L(M) but not in La. (M).

Be aware that when discussing the language the state machine accepts as opposed to the language that provides all of the state machine's actions, it might be beneficial to take language confinement into account. As accepting states are the end state of the machine for any activity in La(M), they are also known as final states. It explores accepting states in more detail. In figure 2 illustrate the trends in renewable integration.

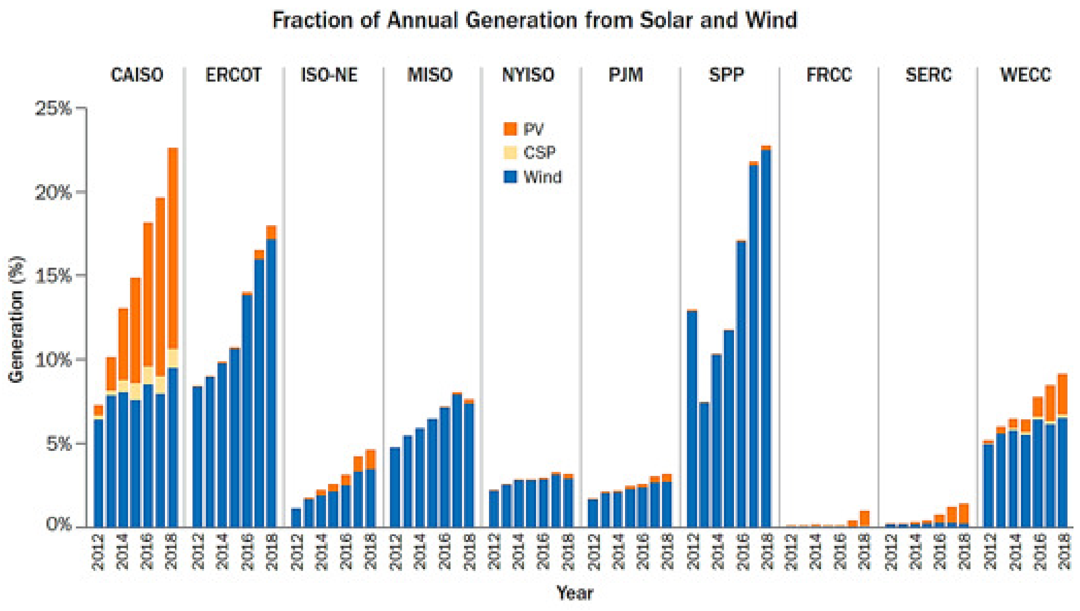


Figure 2: Illustrate the Trends in Renewable Integration.

A language is a collection of value sequences from an alphabet. A regular language is one that is acknowledged by an FSM. Sequences of the pattern 0 n1 n, or a series of n zeros followed by n ones, are a typical illustration of a language that is not regular. The fact that a finite state machine would need to count the zeros to verify that the number of ones matched makes it clear that no such machine can understand this language. And there are an unlimited amount of zeros. On the other hand, the input sequences of the type 10101 01 that the FSM accepts in the box on page 384 are regular.

The notation for describing regular languages is called a regular expression. The Kleene star (also known as the Kleene closure), which bears the name of American mathematician Stephen Kleene, is a key component of regular expressions (who pronounced his name KLAY-nee). The set of all finite sequences of elements from a set V is denoted by the notation V, where V is the set.

For instance, the empty sequence (commonly abbreviated) and any finite sequence of zeros and ones are included in the set V if V = 0 and 1, respectively. Sets of sequences may be given the Kleene star treatment. A is the collection of all finite sequences in which zeros and ones always occur in pairs, for instance, if A = 00, 11. This is represented by the regular expression notation (00|11)*, where the vertical bar denotes the word "or." The set A is defined by the text enclosed in parentheses.

Sequences of letters from the alphabet and sets of sequences make up regular expressions. Assume that our alphabet consists of the lowercase letters A, B, C, and Z. Then grey is a regular expression that designates a single four-character sequence. Grey|gray refers to a collection of two sequences. Sequences or collections of sequences may be grouped using parentheses. For instance, gr(ela)y and (grey)|(gray) have the same meaning.

Regular expressions may also be conveniently notated to make them smaller and easier to understand. For instance, the Kleene star denotes "zero or more," but the + operator denotes "one or more." For instance, a+ is the same as a(a*) and defines the sequences a, aa, aaa, etc. "Zero or one" is a species of the operator? For instance, colou?r designates a set comprising the two sequences colour and colour, and is equivalent to colo(|u)r, where stands for the empty sequence. Software systems often employ regular expressions for pattern matching.

In comparison to the ones shown above, a typical implementation offers many additional convenience notations. Nevertheless, Machine M3 has additional behaviours. With the identical inputs, it can create every output sequence that M1 and M2 can produce in addition to additional outputs. Hence, M1 and M2 are both linguistic improvements of M3. A sound abstraction in relation to LTL formulae concerning input and output sequences is ensured by language confinement. This means that every LTL formula concerning inputs and outputs that holds for B also holds for A if A is a language refinement of B.

Only finite sequences are present in the regular languages covered in the boxes on pages 384 and 385. Yet, limitless executions are a typical feature of embedded systems. We may utilise a Buchi automaton, which bears the name of Swiss logician and mathematician Julius Richard Buchi, to expand the concept of regular languages to infinite runs. A B-uchi automaton is an FSM with one or more acceptable states and the potential to be nondeterministic. The set of behaviours satisfying the LTL formula GF(s1 sn), where s1, , and sn are the accepting states, is referred to as the language accepted by the FSM. These behaviours visit one or more of the accepting states infinitely frequently. An omega-regular language, also known as a -regular language, is a generalisation of regular languages. Because is used to create endless sequences, as described in the box on page 500, it makes sense to utilise in the name.

Several model checking queries may be described by providing a Buchi automaton and then determining if the -regular language it defines includes any sequences. In the part after this one, we'll show that language confinement is flawed when it comes to LTL formulae that make references to the states of state machines. An LTL formula that relates to the states of one machine may not even be applicable to the other machine since language confinement does not mandate that the states of the state machines match. Simulation is necessary to create a sound abstraction that references states.

Trace containment a word that often alludes to language containment here solely refers to the visible trail and not the execution trace. As we'll see in a moment, things get considerably more nuanced when taking execution traces into account. Even though the language between two nondeterministic FSMs is the same, they may behave differently in specific contexts. Linguistic equivalence only asserts that the two machines are capable of creating the identical sequences of output values when given the same sequences of input valuations. Yet as they operate, they take decisions made possible by nondeterminism. These decisions might lead to one of the machines reaching a point where it is no longer able to match the outputs of the other without the ability to see into the future.

Each machine is free to use any policy to decide when presented with a nondeterministic option. Assume that the machine is unable to predict the future, both in terms of future inputs and future decisions that other machines will make. The ability of each machine to make decisions that allow it to match the response of the other machine (producing the same outputs) and further enable it to continue doing such matching in the future is a requirement for two machines to be equivalent. Apparently, language equivalence is insufficient to guarantee that this is possible.

Consider creating separate copies of the environment that accepts M2 for each of the two machines. M1 must take the transition to state b and generate the output y = 0 in the first reaction where x is present. M2 must decide between f and h, though. M2 matches the output y = 0 of M1 regardless of the decision it makes, but it eventually reaches a point where it is no longer able to consistently match M1's outputs. In the second reaction where x is present, M1 can choose a transition that M2 can never match if it is able to observe the state of M2 at the time of its decision. With such a policy in place for M1, it is guaranteed that M1 will never behave in the same way as M2 under the same inputs. Therefore, it is risky to switch M2 for M1.

CONCLUSION

Power and energy analysis is a vital field of study that plays a critical role in ensuring the efficient and sustainable use of energy resources. Through the measurement and analysis of energy consumption and efficiency, and the design and optimization of energy-efficient systems, power and energy analysis helps to identify opportunities for reducing energy waste and improving energy performance. In today's world, where the demand for energy is increasing rapidly and concerns around climate change and energy security are growing, power and energy analysis has become an increasingly important area of research and development. By enabling the development of more efficient and sustainable energy systems, power and energy analysis can help to reduce the environmental impact of energy production and consumption, while also supporting economic growth and development.

REFERENCES

[1] I. Petkov and P. Gabrielli, “Power-to-hydrogen as seasonal energy storage: an uncertainty analysis for optimal design of low-carbon multi-energy systems,” *Appl. Energy*, 2020, doi: 10.1016/j.apenergy.2020.115197.

[2] M. E. Fouda, A. S. Elwakil, A. G. Radwan, and A. Allagui, “Power and energy analysis of fractional-order electrical energy storage devices,” *Energy*, 2016, doi: 10.1016/j.energy.2016.05.104.

[3] A. Cailliau and A. Van Lamsweerde, “Runtime monitoring and resolution of probabilistic obstacles to system goals,” *ACM Trans. Auton. Adapt. Syst.*, 2019, doi: 10.1145/3337800.

[4] I. M. Albayati, A. Postnikov, S. Pearson, R. Bickerton, A. Zolotas, and C. Bingham, “Power and energy analysis for a commercial retail refrigeration system responding to a static demand side response,” *Int. J. Electr. Power Energy Syst.*, 2020, doi: 10.1016/j.ijepes.2019.105645.

[5] H. Wang, Z. Yan, X. Xu, and K. He, “Probabilistic power flow analysis of microgrid with renewable energy,” *Int. J. Electr. Power Energy Syst.*, 2020, doi: 10.1016/j.ijepes.2019.105393.

[6] W. G. Ali and S. W. Ibrahim, “Power Analysis for Piezoelectric Energy Harvester,” *Energy Power Eng.*, 2012, doi: 10.4236/epe.2012.46063.

[7] J. Devlin, K. Li, P. Higgins, and A. Foley, “A multi vector energy analysis for interconnected power and gas systems,” *Appl. Energy*, 2017, doi: 10.1016/j.apenergy.2016.08.040.

[8] M. E. M. Diouri *et al.*, “Assessing power monitoring approaches for energy and power analysis of computers,” *Sustain. Comput. Informatics Syst.*, 2014, doi: 10.1016/j.suscom.2014.03.006.

CHAPTER 21

SEQUENTIAL SOFTWARE IN A CONCURRENT WORLD

Mr. Anil Agarwal, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-anil.agarwal@jnujaipur.ac.in

ABSTRACT:

Sequential software in a concurrent world refers to the study of how traditional sequential programming approaches can be adapted and optimized to work efficiently in the context of concurrent systems. In a world where computer systems are becoming increasingly parallelized and distributed, there is a growing need for software that can execute in a concurrent, multi-threaded environment. This field of study involves the exploration of different programming models and paradigms, such as functional programming and actor-based programming that can enable the development of software that is better suited to concurrency. It also involves the use of specialized tools and libraries that can simplify the task of developing concurrent software, such as lock-free data structures and concurrent collections.

KEYWORDS:

Concurrent System, Data Structure, Sequential Software, Programming, Specialized Tool.

INTRODUCTION

On the other hand, is it safe for M2 to replace M1 if M1 is acceptable in some environments? What it means for M1 to be acceptable in the environment is \sthat whatever decisions it makes are acceptable. Thus, in the second reaction \swhere x is present, both outputs y = 1 and y = 0 are acceptable. In this second \sreaction, M2 has no choice but to produce one or the other these outputs, and it \swill inevitably transition to a state where it continues to match the outputs of M1 \s(henceforth forever absent) (henceforth forever absent). Hence it is safe for M2 to replace M1[1], [2].

In the above example, we can think of the machines as maliciously trying to make M1 \slook different from M2. Since they are free to use any policy to make choices, they are \sfree to use policies that are contrary to our goal to replace M2 with M1. Note that the machines do not need to know the future; it is sufficient to simply have good visibility of the \spresent. The question that we address in this section is: under what circumstances can we \sassure that there is no policy for making nondeterministic choices that can make machine \sM1 observably different from M2? The answer is a stronger form of equivalence called \sbisimulation and a refinement relation called simulation. We begin with the simulation \srelation. \s388 Introduction to Embedded Systems, Lee & Seshia

Simulation is defined by a matching game. To determine whether M1 simulates M2, \swe play a game where M2 gets to move first in each round. The game starts with both \smachines in their initial states. M2 moves first by reacting to an input valuation. If this \sinvolves a nondeterministic choice, then it is allowed to make any choice. Whatever it \schoses, an output valuation results and M2’s turn is over[3], [4].

It is now M1’s turn to move. It must react to the same input valuation that M2 reacted \sto. If this involves a nondeterministic choice, then it must make a choice that matches \sthe output valuation of M2. If there are multiple such choices, it must select one without \sknowledge of the future inputs or future moves of M2. Its strategy should be to choose \sone that enables it to continue to match M2, regardless of what future inputs arrive or \sfuture decisions M2 makes.

Machine M1 “wins” this matching game (M1 simulates M2) if it can always match the \soutput symbol of machine M2 for all possible input sequences. If in any reaction M2 can \sproduce an output symbol that M1 cannot match, then M1 does not simulate M2.

It's interesting to note that all games across all inputs may be compactly recorded if M1 replicates M2. Let S1 and S2 represent the states of M1 and M2, respectively. The collection of pairings of states that the two machines inhabited in each round of the game for all potential inputs is known as a simulation relation S S2 S1. The game's probable plays are included in this set.

Secondly, take note that the connection includes the states of the two machines in the first round since it contains the pair (e, a) of beginning states. M1 will be in b and M2 might be in either f or h in the second round. These two scenarios are also taken into consideration. M1 will be in c or d after the third round, while M2 will be in either g or i.

If a simulation connection covers all game scenarios, it is comprehensive. M2, the machine that moves first, has unconstrained movements, therefore it must take into consideration all of its attainable states. It is not required to take into consideration all of M1's attainable states since its movements are limited by the need to match M2. We may explicitly describe a simulation connection using the formal model of nondeterministic FSMs provided. Let M1 = (States1, Inputs, Outputs, PossibleUpdates1, InitialState1) and M2 = (States2, Inputs, Outputs, PossibleUpdates2, InitialState2) be the first and second models, respectively[5], [6].

Suppose that the two machines are of the same kind. Both machines' possibleUpdates functions always return a set with a single element if they are deterministic. The simulation relation is presented as a subset of States2 States1 if M1 simulates M2. The machine that moves first in the game, M2, the one that is being replicated, is first in States2 States1. Take note of the ordering here. To analyse the opposite case, the relation is defined as a subset of States1 States2 if M2 mimics M1. M1 has to move first in this iteration of the game.

The "winning" tactic may be expressed numerically. If there is a subset S States2 States1 such that 1. (initialState2, initialState1) S, and Lee & Seshia, Introduction to Embedded Systems 391, we say that M1 replicates M2.

The simulation relation is the name given to this set S, if it exists. It creates a relationship between the two machines' states. If it doesn't, M1 can't replicate M2 if it doesn't exist.

Assuming that M1 simulates M2 and M2 simulates M3, then M1 simulates M3 since simulation is transitive. For example, if simulation relations S2,1 States2 States1 (M1 simulates M2) and S3,2 States3 States2 (M2 simulates M3) are supplied, then S3,1 = (s3, s1) States3 States1 | there exists s2 States2 where (s3, s2) S3,2 and (s2, s1) S2,1.

It is simple to verify that M1 replicates M2. Keep in mind that M1 is nondeterministic and has two unique methods in which it might match the movements of M2 in two of its states[7], [8]. It may haphazardly choose one of these options to fit the motions. If it always opts to go back to state a from state b, then the simulation relation is S2,1 = (ac, a),(bd, b).

DISCUSSION

Simulation is often used to connect a less complex specification M1 to a more complex realisation M2, as is the case with all abstraction-refinement interactions. The language of M1 includes the language of M2 when M1 replicates M2, yet the guarantee is more reliable than language containment. The following theorem provides an overview of this phenomenon.

Let S represent the simulation connection. Identify all M2 ((x0, s0, y0), (x1, s1, y1), (x2, s2, y2), and behavior-producing execution traces (x, y). (There will only be one execution trace if M2 is deterministic.) The simulation relation guarantees that we will be able to locate an execution trace for M1 ((x0, s0 0, y0),(x1, s0 1, y1),(x2, s0 2, y2),), where (si, s0 I) S, such that, given input value xi, M1 outputs yi. So (x, y) L. (M1). This theorem may be used to demonstrate that M1 does not imitate M2 by demonstrating that M2 exhibits characteristics that M1 does not.

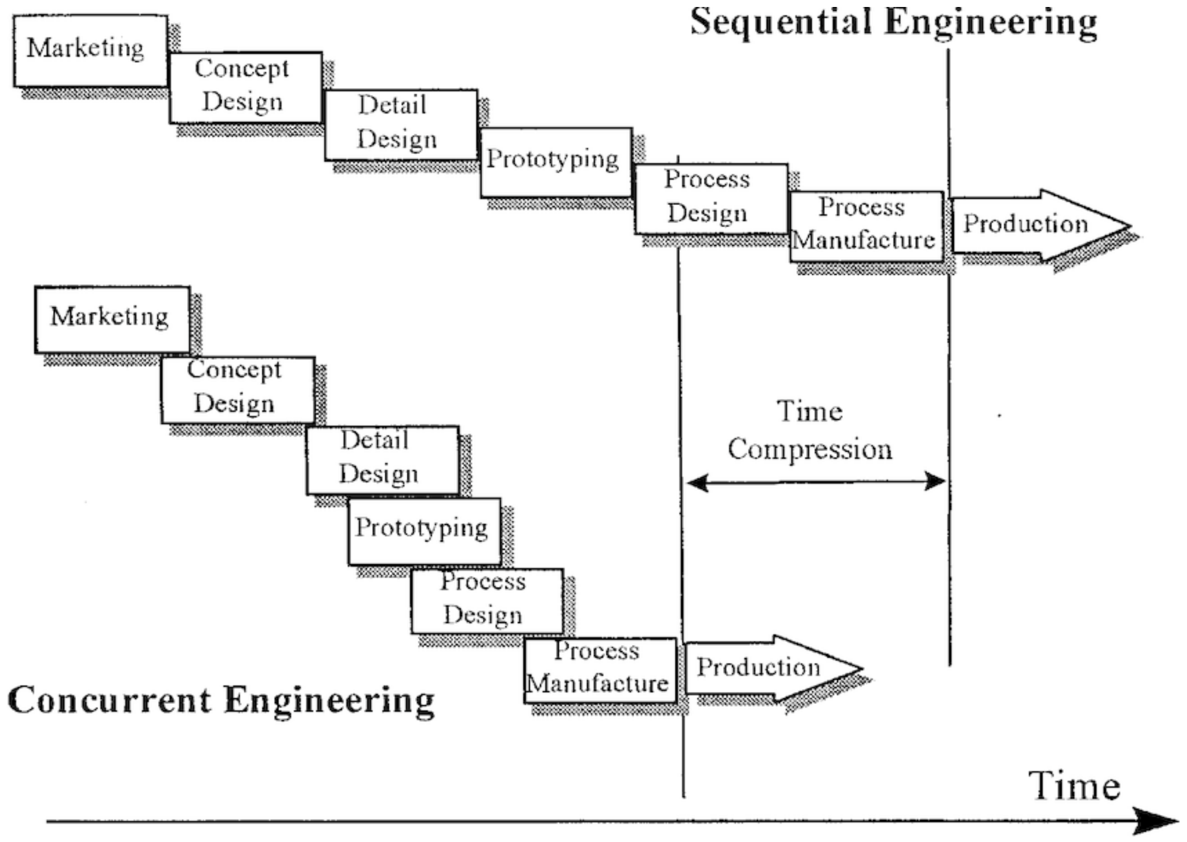


Figure 1: Illustrate the concurrent and sequential engineering.

Understanding what the theorem says and what it does not say is crucial. For instance, it does not state that M1 mimics M2 if L(M2) L(M1), this claim is untrue. The languages of these two devices are identical. Despite the fact that the input/output behaviours of the two machines are the same, there are observable differences between them.

Naturally, if M1 and M2 are deterministic and M1 simulates M2, then M2 simulates M1 and their languages are the same. Hence, only for nondeterministic FSMs does the simulation relation depart from language confinement. Even though M1 and M2 are two distinct machines that M1 mimics and M2 simulates, it is feasible for this to happen. It should be noted that these two machines' languages must match according to the theory in the preceding section.

The following describes how nondeterministic decisions for two machines function if M1 mimics M2 and M2 simulates M1, but they are not bisimilar. They toss a coin to determine which machine moves first in each response. The computer chooses a move based on an input value. The second machine must be able to match all of its potential options. In this situation, the machines may reach a point when neither machine can match all of the potential motions of the other. In Figure 1 illustrate the concurrent and sequential engineering.

Let's say, for example, that M2 gets to move first during the opening move. M1 will need to match each of its three potential movements. Let's say it decides to go to f or h. If M1 gets to move first in the next round, M2 will no longer be able to match all of its potential movements. It should be noted that this argument does not invalidate the finding that these machines replicate one another. M1 will always be able to counteract M2's movements if it moves first in every round. Similarly, if M1 chooses to move first in every round, M2 may always equal it by deciding to go to j in the first round. The ability to switch which machinery move first causes the discernible difference.

We want a higher equivalence relation called bisimulation in order to guarantee that two machines are observably equal in all settings. If we can play the modified matching game where either machine may move first in each round, we can state that M1 and M2 are bisimilar to one another (or that M1 bisimulates M2). Let M1 = (States1, Inputs, Outputs, possibleUpdates1, initialState1) and M2 = (States2, Inputs, Outputs, possibleUpdates2, initialState2) be the formal model of nondeterministic FSMs used to describe the bisimulation relation.

Suppose that the two machines are of the same kind. Both machines' possibleUpdates functions always return a set with a single element if they are deterministic. The simulation relation is presented as a subset of States2 States1 if M1 bisimulates M2. As long as M1 bisimulates M2, M2 will also bisimulate M1, therefore the ordering in this case is irrelevant. In today's computing landscape, concurrency has become an essential aspect of software development. With the proliferation of multi-core processors, distributed systems, and cloud computing, software developers are increasingly tasked with designing applications that can efficiently execute multiple tasks concurrently.

Despite this trend towards concurrency, much of the software in use today is still designed using sequential programming techniques. Sequential programming is a traditional programming paradigm where programs execute one instruction at a time, in a predetermined order. In this paradigm, a program's execution is a linear sequence of steps, with each step depending on the previous one.

In contrast, concurrent programming is a programming paradigm where programs execute multiple tasks concurrently, either on a single processor or across multiple processors. Concurrent programming involves designing programs to take advantage of multiple processors and to manage the synchronization and communication between multiple tasks.

We will explore the role of sequential software in a concurrent world. We will examine the reasons why sequential software is still prevalent and discuss the challenges of transitioning to concurrent programming. We will also explore some of the tools and techniques that can be used to design and implement concurrent software.

Why is sequential software still prevalent?

Despite the growing importance of concurrent programming, much of the software in use today is still designed using sequential programming techniques. There are several reasons why sequential programming is still prevalent.

Familiarity and simplicity

Sequential programming is a well-established programming paradigm that most software developers are familiar with. Many developers have years of experience writing sequential programs, and they find it easier to design and implement software in this paradigm. Sequential programs are easier to reason about and debug, and the behavior of the program is more predictable.

Legacy systems

Many software systems that are in use today were designed using sequential programming techniques. These systems are often critical to the operation of businesses and organizations and have been in use for many years. Replacing these systems with new, concurrent systems can be expensive and time-consuming, and there is often a reluctance to invest in such a large undertaking.

Single-threaded nature of many applications

Many applications are naturally single-threaded, meaning that they only need to execute one task at a time. For example, a word processor only needs to execute one command at a time, and a web server only needs to process one request at a time. For such applications, there is little benefit to using a concurrent programming paradigm.

Performance

Sequential programs can still be more performant than concurrent programs in some cases. This is particularly true for applications that have a small number of tasks that need to be executed. In such cases, the overhead of managing concurrency can outweigh the benefits of using multiple processors.

Challenges of transitioning to concurrent programming

Transitioning from sequential programming to concurrent programming can be challenging. There are several challenges that must be overcome.

Complexity

Concurrent programming is inherently more complex than sequential programming. Concurrent programs must manage multiple tasks that can execute concurrently and must coordinate the communication and synchronization between these tasks. This added complexity can make it more challenging to design, implement, and debug concurrent programs.

Race conditions

In concurrent programs, race conditions can occur when two or more tasks try to access the same shared resource simultaneously. These race conditions can lead to unpredictable behavior and can be difficult to detect and debug. To avoid race conditions, concurrent programs must use synchronization techniques such as locks, semaphores, and monitors.

Deadlocks

Deadlocks can occur in concurrent programs when two or more tasks are waiting for each other to release a resource. Deadlocks can cause the program to hang and can be difficult to detect and debug. To avoid deadlocks, concurrent programs must use techniques such as resource ordering and timeout mechanisms.

Performance

Concurrency can improve the performance of a program by allowing multiple tasks to execute concurrently. However, the overhead of managing concurrency can also degrade performance. The advent of multi-core processors and distributed systems has made parallel and concurrent processing a necessity for modern software development. However, sequential programming still has a significant role to play in software development. This article will explore the relationship between sequential software and concurrent software and how they coexist in today's world.

Sequential Software:

Sequential software is a program that executes instructions in a linear sequence, one after the other. This type of program is straightforward to write and understand, making it easy to debug and maintain. Sequential programs are typically used for small and simple tasks that do not require complex computations or long-running processes.

Sequential software is prevalent in areas such as desktop applications, file processing, and simple web applications. For example, a text editor or a calculator program is typically implemented as a sequential program. These types of programs have a single thread of execution, and the user interacts with the program on a one-to-one basis.

However, sequential software has some significant limitations. One of the most significant drawbacks of sequential software is its inability to scale. As the program's complexity increases, its execution time also increases linearly. This makes it challenging to handle large-scale applications, such as big data processing, machine learning, and high-performance computing.

Concurrent Software:

Concurrent software is a program that executes multiple tasks simultaneously. This type of program is designed to take advantage of modern computer hardware's capabilities, such as multi-core processors and distributed systems. Concurrent programs typically have multiple threads of execution that run independently and share resources.

Concurrent software is suitable for applications that require parallel processing of large data sets, such as video and image processing, financial analysis, and scientific simulations. Concurrent programming is also used in real-time systems such as robotics and gaming, where the system must respond to events in real-time.

Concurrency provides many benefits, such as increased performance, improved responsiveness, and scalability. However, concurrent programming is more challenging than sequential programming. Concurrent programs must deal with synchronization and communication issues, such as race conditions, deadlocks, and livelocks. These issues can lead to bugs that are difficult to diagnose and fix.

Sequential vs. Concurrent Programming:

Sequential and concurrent programming have different strengths and weaknesses, and each has its place in software development. Sequential programming is simple to write and understand, making it ideal for small and straightforward applications. Concurrent programming is more challenging to write and debug, but it provides superior performance and scalability.

The choice between sequential and concurrent programming depends on the application's requirements. Applications that require high performance, scalability, and responsiveness typically require concurrent programming. On the other hand, applications that are simple and require only a small amount of processing power can be implemented using sequential programming.

Concurrency Models:

There are two main concurrency models used in concurrent programming: shared-memory and message-passing.

Shared-Memory Concurrency:

In shared-memory concurrency, multiple threads of execution share the same memory space. Threads can access and modify shared variables, which can lead to synchronization issues such as race conditions, deadlocks, and livelocks. To prevent synchronization issues, shared-memory concurrency requires the use of synchronization primitives such as locks, semaphores, and monitors. These primitives provide a way to ensure that only one thread can access a shared resource at a time.

Message-Passing Concurrency:

In message-passing concurrency, threads of execution communicate by sending and receiving messages. Each thread has its own private memory space, and communication between threads is achieved through message passing.

Message-passing concurrency is typically more scalable than shared-memory concurrency since threads do not need to access shared resources. However, message-passing concurrency can be more challenging to implement since it requires more communication overhead.

Concurrency in Practice:

Concurrency is an essential aspect of modern software development, and many programming languages and frameworks support concurrent programming

CONCLUSION

While concurrent programming is becoming increasingly important in today's computing landscape, sequential programming remains prevalent for several reasons. Sequential programming is familiar and simple, many legacy systems were designed using sequential programming techniques, and many applications are naturally single-threaded. However, transitioning to concurrent programming can bring benefits such as improved performance and scalability. The transition to concurrent programming is not without its challenges, however. Concurrent programming is more complex than sequential programming, and concurrency can lead to race conditions and deadlocks. To overcome these challenges, developers must use synchronization techniques such as locks, semaphores, and monitors, and employ resource ordering and timeout mechanisms to avoid deadlocks.

REFRENCES

[1] T. Yu, W. Wen, X. Han, and J. H. Hayes, "ConPredictor: Concurrency Defect Prediction in Real-World Applications," *IEEE Trans. Softw. Eng.*, 2019, doi: 10.1109/TSE.2018.2791521.

[2] T. Yu, Z. Huang, and C. Wang, "ConTesa: Directed test suite augmentation for concurrent software," *IEEE Trans. Softw. Eng.*, 2020, doi: 10.1109/TSE.2018.2861392.

[3] M. Vujošević Janičić, "Concurrent Bug Finding Based on Bounded Model Checking," *Int. J. Softw. Eng. Knowl. Eng.*, 2020, doi: 10.1142/S0218194020500242.

[4] S. Flur *et al.*, "Mixed-size concurrency: ARM, POWER, C/C++11, and SC," *ACM SIGPLAN Not.*, 2017, doi: 10.1145/3009837.3009839.

[5] R. Meier, A. Rigo, and T. R. Gross, "Virtual machine design for parallel dynamic programming languages," *Proc. ACM Program. Lang.*, 2018, doi: 10.1145/3276479.

[6] R. Ramakrishnan and A. Kaur, "Little's law based validation framework for load testing," *Inf. Softw. Technol.*, 2019, doi: 10.1016/j.infsof.2018.11.007.

[7] J. Huang, P. O. Meredith, and G. Rosu, "Maximal sound predictive race detection with control flow abstraction," *ACM SIGPLAN Not.*, 2014, doi: 10.1145/2666356.2594315.

[8] S. H. Wu, J. Y. H. Fuh, and A. Y. C. Nee, "Concurrent process planning and scheduling in distributed virtual manufacturing," *IIE Trans. (Institute Ind. Eng.*, 2002, doi: 10.1080/07408170208928851.

CHAPTER 22

REACHABILITY ANALYSIS AND MODEL CHECKING

Prof. Sudhir Kumar Sharma, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-hodece_sadtm@jnujaipur.ac.in

ABSTRACT:

Reachability analysis and model checking are two powerful techniques used in formal verification to verify the correctness of software and hardware systems. Reachability analysis involves determining whether a state in a system can be reached from a given initial state, while model checking involves checking a formal model of a system against a set of properties. Both techniques are used to detect errors, bugs, and design flaws in software and hardware systems before they are deployed. Reachability analysis is typically used in the context of model-based testing and verification. It involves exploring the system's state space to determine whether certain states can be reached from a given initial state. Reachability analysis can be performed using various algorithms, such as depth-first search, breadth-first search, and iterative deepening.

KEYWORDS:

Errors, Depth-first search, Reachability Analysis, Model Checking, Software System.

INTRODUCTION

Techniques for formally expressing system features and models, as well as for comparing such models, were covered. The topic of determining whether a system fulfils its formal specification in its designated operating environment will be studied in this chapter along with algorithmic strategies for formal verification. Specifically, we investigate the model verification method. Model checking uses an algorithm to examine whether a system complies with a formal specification that is stated as a temporal formula[1], [2].

The idea of a system's set of attainable states is crucial to model checking. The technique of calculating a system's set of attainable states is known as reachability analysis. In this chapter, reachability analysis and model checking's foundational methods and concepts are presented. Examples from the design of embedded systems, such as the verification of high-level models, sequential and concurrent software, control, and robot route planning, are used to explain these techniques. Model checking is a vast and active area of research, thus a thorough discussion of the topic is outside the purview of this chapter[3].

Any system with no inputs is said to be closed. In contrast, an open system is one that continuously interacts with its environment by taking inputs from it and potentially returning output to it. The model of the closed system M that results from combining the model of the system S that has to be verified with a model of its environment is often subjected to formal verification techniques. E. Typically, S and E are open systems in which S receives all inputs from E and vice versa [4].

In order to determine whether or not S meets the property in environment E, the verifier outputs a YES/NO response. A counterexample, also known as an error trace, which is a trace of the system that shows how is violated, is often provided with a NO output. Counterexamples are excellent tools for debugging. A proof or certificate of correctness that includes a YES response is included in certain formal verification tools; this output may be helpful for certifying the correctness of a system. The manner in which the system model S and environment model E are combined depends on how the system and environment interact. State machine model composition techniques. A verification model M may be created from S and E using any of these techniques of composition. Remember that M may not be deterministic[5].

For the sake of simplicity, we'll assume in this chapter that system composition has already been completed using one of the methods covered. The integrated verification model M will be the operating platform for all algorithms mentioned in the following sections, all of which are focused on determining if M meets property. We show how one may prove a Gp property by calculating the attainable states of a system. We shall discuss a method for reachability analysis of finite-state machines based on an explicit state enumeration. Every trace that a system M is capable of exhibiting must fulfil Gp for a system M to satisfy Gp, where p is a proposition. By listing every state of M and making sure that each one meets p, this property may be confirmed.

Theoretically, such enumeration is always feasible when M is a finite state the state space of M may be represented as a directed graph, with the nodes representing the states of M and the edges representing their transitions. The collection of all states is referred to as its state space, and this graph is known as the state graph of M. From a graph-theoretic perspective, it is clear that verifying Gp for a finite-state system M corresponds to iteratively traversing M's state graph, beginning with the initial state, and verifying that each state encountered along the way satisfies p. This traversal must come to an end since M has a limited number of states [6], [7].

Assume that the environment E of the system S is the pedestrian model and that the system S is the traffic light controller. Let M be the synchronous composition of S and E. As you can see, M is a closed system. Let's say we want to make sure M meets the requirement G (green crossing). In other words, we want to make sure that it never happens that pedestrians cross the street when the signal is green the composite system M as an extended FSM. M has no inputs or outputs, as you can see. 188 states make up the finite-state system M the complete state graph of M, where each node stands for a set of states, one for every variation in count in that node. The LTL condition G (green crossing) is satisfied by every trace in this graph because, as we can see from a visual analysis of the graph, no state meets the proposition (green crossing).

In reality, it is more difficult than in the preceding example to determine if a finite-state system M fulfils a Gp condition for the following reasons: Usually, just the starting state and transition function are provided, and the state graph must be created as needed. Extended state machine created from synchronous-reactive synthesis of pedestrian and traffic light controller models. The system might contain many more states than the syntactic description of M, perhaps exponentially so. As a result, conventional data structures like an adjacency or incidence matrix are unable to accurately depict the state graph [8].

In this part, we go through how to generate and traverse the state graph on-the-fly in order to calculate the accessible state set. First, keep in mind that the system of interest M might be nondeterministic, closed, and finite-state. The collection of potential future states for M depends only on its present state since it has no inputs.

DISSCUSION

Given M's starting state s0 and transition relation, algorithm 15.1 computes the set of states that M is capable of reaching. Beginning with state s0, procedure DFS Search traverses the state graph of M in depth first. By repeatedly applying to the states visited throughout the traverse, the graph is instantly produced.

The stack holding the current route in the state graph being explored from s0 and R, the current collection of states accessed during traversal, are the two basic data structures needed by the method. As M has a limited number of states, all states that can be reached from s0 will eventually be in R. This means that no additional states will be put onto M, which means that M will eventually become empty. The set of all attainable states of M is what R represents when the method DFS Search comes to a finish.

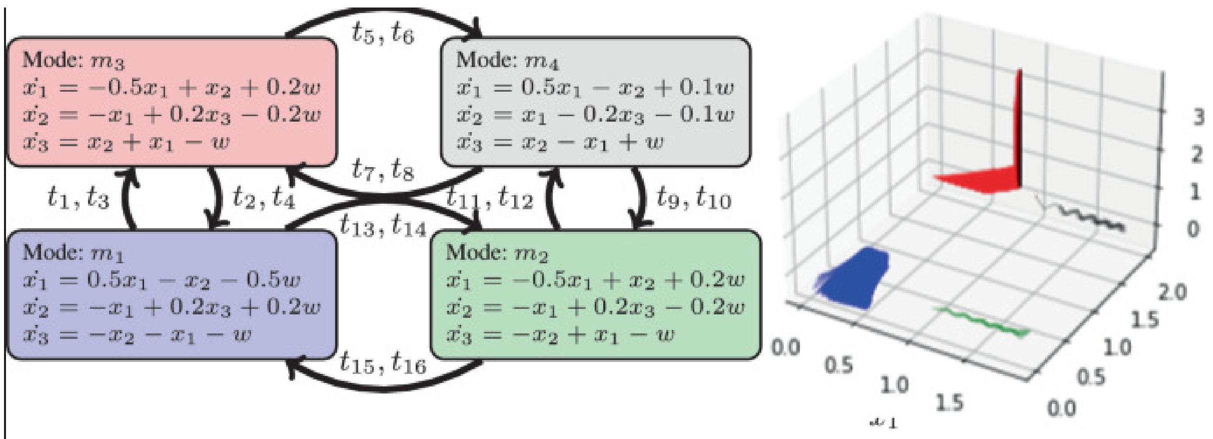
This algorithm's space and time requirements scale linearly with the size of the state network (see Appendix B for an introduction to such complexity notions). Yet, the size of the descriptions of S and E might exponentially increase the number of nodes and edges in the state graph of M. For instance, if S and E contain 100 Boolean state variables jointly (a modest 410 Introduction to Embedded Systems, Lee & Seshia

The state graph of M may contain a total of 2 100 states, which is far more than what modern computers can store in main memory. 15. REACHABILITY ANALYSIS AND MODEL CHECKING Consequently, state compression methods must be included to explicit-state search algorithms like DFS Search.

Remember that the state space of a composition of k finite-state systems is the cartesian product of the state spaces of M1, M2,..., Mk (say, using synchronous composition). In other words, the composition of M1, M2,..., and Mk may have k i=1ni states if M1, M2,..., and Mk, respectively, have n1, n2,..., and nk states. The exponential growth in the number of states for a concurrent composition of k components is evident. Scaling does not occur when the state space of the composite system is explicitly represented. We will provide methods that, in certain circumstances, may reduce this issue in the next section.

A set of states should be symbolically represented as a propositional logic formula rather than directly as a collection of distinct states. This is the fundamental concept behind symbolic model checking. Such formulae are often represented and operated on effectively using specialised data structures. Hence, symbolic model checking acts on groups of states as opposed to explicit-state model checking, which manipulates individual states.

A closed, finite-state system M has a symbolic algorithm known as algorithm 15.2 (Symbolic Search) that calculates the set of attainable states. The input-output specifications for this method are the same as those for the prior explicit-state algorithm DFS Search, but all operations in Symbolic Search are set operations. In Figure 1 illustrate the cyber-physical system analysis.


Mode: m_3
$\dot{x_1} = -0.5x_1 + x_2 + 0.2w$
$\dot{x_2} = -x_1 + 0.2x_3 - 0.2w$
$\dot{x_3} = x_2 + x_1 - w$
Mode: m_4
$\dot{x_1} = 0.5x_1 - x_2 + 0.1w$
$\dot{x_2} = x_1 - 0.2x_3 - 0.1w$
$\dot{x_3} = x_2 - x_1 + w$
Mode: m_1
$\dot{x_1} = 0.5x_1 - x_2 - 0.5w$
$\dot{x_2} = -x_1 + 0.2x_3 + 0.2w$
$\dot{x_3} = -x_2 - x_1 - w$
Mode: m_2
$\dot{x_1} = -0.5x_1 + x_2 + 0.2w$
$\dot{x_2} = -x_1 + 0.2x_3 - 0.2w$
$\dot{x_3} = -x_2 + x_1 - w$
t_5, t_6
t_7, t_8
t_1, t_3
t_2, t_4
t_{11}, t_{12}
t_9, t_{10}
t_{13}, t_{14}
t_{15}, t_{16}
x_1

IDs	Guard	Reset	IDs	Guard	Reset
t_1, t_{11}	$x_2 \geq 1$	none	t_2, t_9	$x_2 \leq 1$	none
t_3, t_{12}	$x_2 \leq 0$	$x_2 := 1.9, x_3 := 0$	t_4, t_{10}	$x_2 \geq 2$	$x_2 := 0.1, x_3 := 0$
t_5, t_{13}	$x_1 \geq 1$	none	t_7, t_{15}	$x_1 \leq 1$	none
t_6, t_{14}	$x_1 \leq 0$	$x_1 := 1.9, x_3 := 0$	t_8, t_{16}	$x_1 \geq 2$	$x_1 := 0.1, x_3 := 0$

Figure 1: Illustrate the Reachability Analysis for Cyber-Physical Systems.

The full collection of states reached at any stage in the search are represented by R in the algorithm Symbolic Search, and the new states created at that time are represented by Rnew. The process ends when no new states can be formed, and R stores all states that can be reached from s0. Line where Rnew is calculated as the set of all states s 0 accessible from any state s in R in one step of the transition relation, is the most important phase in the procedure.

Given that it includes calculating the function's image, this procedure is known as a "image computation." Symbolic reachability algorithms are based on effective implementations of image computation that directly work on propositional logic formulations. The main set operations in Symbolic Search, in addition to image processing, are set union and emptiness testing.

We need to introduce some notation first. Allow vl to represent the traffic light controller FSM S's state at the beginning of each reaction, i.e., vl = "green, yellow, red, pending". Let crossing, none, and waiting indicate the states of the pedestrian FSM E, respectively. With this notation, the propositional logical formula for the starting state set s0 of the composite system M is as follows:

Such steps are taken until there is no more growth in the set of accessible states R. The ultimate set that may be reached is denoted by the following symbols: vl = red; vp = crossing; 0; count; 60; vl = green; vp; none; 0; count; 60; vl = pending; vp; waiting; 0; count; 60; vl = yellow; vp; waiting; 0; count; 5;

The symbolic representation is really a lot more condensed than the explicit one. The preceding example does a good job of illustrating this as inequalities like 0 count 60 may be used to compactly describe a large number of states. It is possible to create computer programmes that work directly with symbolic representations. There are a few examples of these programmes in the box.

The state-explosion issue has been effectively addressed by symbolic model verification for several types of systems, most notably for hardware models. Even symbolic set representations, however, have the potential to have exponentially more system variables in the worst case scenario.

Working with the most basic system abstraction that will offer the necessary proofs of safety is a difficulty in model checking. Smaller state spaces and faster checking are characteristics of simpler abstractions. Of course, the tricky part is deciding which specifics to leave out of the abstraction.

Depending on the property that has to be validated, a system component may need to be abstracted away. This principle is shown by the example that follows. Let's say we hide all references to the variable count from the model, including any guards referencing it and all changes to it. This will abstract the variable count away from M.

This abstract Mabs has more behaviours than M, as we can see. For instance, even if in the real system M this state must be left within five clock ticks, we may take the self-loop transition from the state (yellow, waiting) and remain there perpetually. Moreover, Mabs are capable of displaying all of M's behaviours. The intriguing thing is that we can demonstrate that Mabs fulfils G (green crossing) even with this approximation. The count value has no bearing on this attribute. Observe that M has 188 states, but Mabs has just 4. As we have to examine far fewer states than for M, reachability analysis on Mabs is much simpler.

An abstraction may be calculated in a variety of ways. Localization reduction, also known as localization abstraction, is one of the straightforward and very helpful ways (Kurshan, 1994). In localization reduction, a subset of state variables are hidden in order to abstract away components of the design model that are unrelated to the property being tested. Hiding a variable is equivalent to allowing it to develop at random. It is the kind of abstraction utilised in Example 15.3 above, where count is given complete freedom to vary and all transitions are performed without regard to count's value.

Take into account the multithreaded software that is provided below. The function randomCall is called by the method lock unlock after executing a loop in which it obtains a lock. Depending on the outcome, it either releases the lock and performs another loop iteration or ends the loop (and then releases the lock). Another loop iteration is about to be carried out. Consider the scenario where the attribute we're looking for is that the code doesn't try to invoke pthread mutex lock twice in a row. Remember how the system may deadlock if a thread gets indefinitely stalled while attempting to acquire a lock in Section 11.2.4? In the example above, if the thread already has lock lock and tries to acquire it once again, this may happen.

If we describe this programme precisely, without any abstraction, then we must also reason about all potential old and new values in addition to the program's current state. In this system, assuming a word size of 32, the size of the state space is approximately 2 32 2 32 n, where 2 32 is the sum of the old and new values and n is the size of the remaining state space. To demonstrate the validity of this programme, it is not essential to analyse the specific values of old and new. For the sake of this illustration, let's assume that our programming language has a boolean type. Moreover assume that the programme is capable of carrying out nondeterministic assignments. Thus, using a C-like syntax, we can create the following abstraction of the original programme, where the Boolean variable b stands in for the predicate old == new.

It is clear that this abstraction only keeps the necessary details to demonstrate that the programme fulfils the required characteristic. Since the loop is only iterated if b is set to false, which suggests that the lock was released before the next attempt to acquire, the lock will specifically not be acquired twice.

Observe further that the extent of the state space to be investigated is now only 2n. Here is how utilising the "correct" abstraction may have an impact. Automating the computation of basic abstractions is a significant difficulty for formal verification. Counterexample-guided abstraction refinement (CEGAR), initially proposed by Clarke et al., is a powerful and often used method (2000). While utilising localization reduction, the fundamental notion is to begin by concealing practically all state variables except from those that are referred to by the temporal logic feature. There will be more abstract systems as a consequence.

. As a result, the original system also satisfies the LTL formula if this abstract system does (i.e., each of its behaviours fulfils). Nevertheless, the model checker gives a counterexample if the abstract system does not meet. The procedure ends after discovering a valid counterexample if this counterexample is one for the original system. Otherwise, the CEGAR method examines this counterexample to determine which hidden variables must be made accessible, and then computes an abstraction again using these new variables. The procedure continues until it either produces a legitimate counterexample for the original system or an abstract system is shown to be accurate. Progress in software model verification has been greatly aided by the CEGAR strategy and numerous related approaches. On page 424, in the sidebar, we go over some of the most important concepts. Model Validation of Liveness Parameters

We have limited our verification efforts to features of the kind Gp, where p denotes an atomic proposition. A very limited kind of safety feature is the claim that Gp holds for all traces. Some beneficial system qualities, are not safety properties, however. For instance, the liveness condition "the robot must visit place A" asserts that Fq must hold for all traces if proposition q is a representation of the robot visiting location A. In reality, a number of issues, including robotics route planning issues and the progress characteristics of distributed and concurrent systems, may be characterised as liveness features. To handle this class of characteristics, model checking should be expanded.

Notwithstanding liveness qualities, properties of the type Fp may be partly tested using the methods covered previously in this chapter. Remember from Chapter 13 that if and only if Gp holds for the same trace, then Fp holds for that trace. In other words, if "p is always false," then "p is true some point in the future." We may thus try to confirm that the system fulfils Gp. We know that Fp does not hold for any trace if the verifier claims that Gp holds for all traces. On the other hand, if the verifier returns "NO," the following counterexample serves as a witness illustrating how p could finally turn out to be true. While this witness offers a single trace for which Fp is valid, it does not demonstrate that Fp is valid for all traces (unless the machine is deterministic).

A more advanced strategy is required for inspections that are more thorough and liveness attributes that are more complex. In a nutshell, the following is one method for explicit-state model testing of LTL properties:

1. Create an automaton B to represent the property's negation, with certain of its states designated as accepting states.

2. Create the system automaton M's and the property automaton B's synchronous composition. The product automaton MB experiences accepting states as a result of the property automaton's accepting states.
3. M does not fulfil if the product automaton MB may return to an accepting state infinity of times; otherwise, M does.

The above method of verification is referred to as the automata-theoretic method. In the next part, we provide a succinct introduction to this topic. The foundational works on this subject (Wolper et al. (1983); Vardi and Wolper (1986)) as well as the book on the SPIN model checker (Holzmann (2004)) include further information.

Consider starting with the idea that properties are automata. Consider the information about omegaregular languages that was first presented in the box on page 386. For model verifying liveness qualities, the theory of Buchi automata and omega-regular languages, which was briefly discussed there, is important. A collection of actions that fulfil an LTL property roughly corresponds to an LTL property. The language of the Buchi automaton that corresponds to is made up of this collection of actions. If is the condition that the system must meet for the LTL model checking technique we describe, then we express its negation as a Buchi automaton. Here, we provide a few exemplary instances.

Assume that an FSM M1 simulates a system that runs indefinitely and generates the pure output h (for heartbeat), and that it must generate this output at least once every three reactions. In other words, if it doesn't create the output h in two consecutive reactions, it must in the third. Let's investigate this machine. Any behaviours that enter and remain in state d are included in the language this automaton accepts. In other words, any actions that result in a current output on f in some response are included in the language. The language that the composite machine accepts is empty if, after composing the aforementioned machine with M1, it can never yield f = present. The heartbeat h is produced by M at least once per three reactions if we can demonstrate that the language is empty.

It should be noted that the property 1 in the aforementioned example is a safety property. Here, we provide an example of a liveness attribute. Assume the FSM M2 simulates a controller for a robot that has to find a room and remain there indefinitely. Let p represent the claim that is shown to be accurate when the robot enters the target room. In LTL, the required property 2 may thus be stated as FGp. This property's negation is GFp. The following is the Buchi automaton B2 that corresponds to this negated property: Keep in mind that all of B2's acceptable behaviours line up with those in which p holds every time. These behaviours fit into a cycle where state b of B2 is regularly visited in the state graph for the product automaton. An acceptance cycle is the name given to this cycle.

The requirements of the type G Fp also naturally appear as liveness attributes. As seen in the example below, this kind of property is important for asserting fairness claims, which claim that certain desirable features hold infinitely many times.

Take into account a traffic signal system similar to Example 3.10's. We could want to claim that during any execution, the traffic signal turns green an endless number of times. This means that the state green gets visited an infinite number of times, which is denoted by the expression 3 = GF green.

The automata for 3 is the same as the one for the negation of 2 in Example 15.6 above, using green in place of p. But, in this instance, the automaton's welcoming actions are what are wanted.

The workhorse of explicit-state model testing of LTL qualities is therefore evident from these examples: determining whether a specific accepted state s in an FSM can be visited endlessly often. The algorithm for this issue is then shown. In other words, we are looking for an algorithm to determine if state sa is accessible from itself and from the beginning state s0 of M. You should be aware that asking if a state may be visited infinitely often differs from asking whether it must be visited infinitely frequently.

Like in the instance the graph-theoretic perspective is beneficial in this scenario as well. Let us assume for argument's sake that we have generated the full state graph a priori. So, determining whether state sa is accessible from state s0 is just a graph traversal issue that can be resolved, for instance, by depth-first search (DFS). Also, the issue of determining if sa is accessible from itself equates to determining whether the state graph that contains that state has a cycle.

We must do this search instantly and deal with large state spaces, which are comparable to the primary issues outlined. This issue is resolved by the nested depth-first search (nested DFS) technique, which is implemented in the SPIN model checker (Holzmann, 2004). 15.3. As shown in the Main function at the bottom, the algorithm starts by using the Nested DFS Search method with argument 1. The closed system MB is created by combining the automata B, which represents the negation of the LTL formula, with the initial closed system M.

The concept is to do two depth-first searches, one within the other, as the name implies. A route from the beginning state s0 to the target accepted state sa is identified by the first DFS.

After that, we launch another DFS from SA to see if we can get back to SA. Depending on whether we're doing the first DFS or the second, the variable mode is either 1 or 2. The searches carried out in modes 1 and 2 employ stacks 1 and 2, respectively. The method creates as output the route going from s0 to sa with a loop on sa if sa is discovered in the second DFS. Just reading the contents of stack 1 yields the route from s0 to sa. The cycle from sa to itself is also discovered from stack 2.

This chapter provides several fundamental formal verification procedures, such as model checking—a method for determining if a finite-state system meets a temporal logic condition. Closed systems, which are created by combining a system with its operational environment, provide the basis for verification. Reachability analysis, which validates Gp form qualities, is the first crucial idea. This chapter also covers the abstraction idea, which is essential to the scalability of model verification. This chapter also demonstrates how liveness qualities may be handled using explicit-state model checking methods, where a key idea is the relationship between properties and automata.

CONCLUSION

Reachability analysis and model checking are powerful techniques used in formal verification to verify the correctness of software and hardware systems. These techniques are used to detect errors, bugs, and design flaws in systems before they are deployed, ultimately improving system reliability and safety. Reachability analysis involves exploring a system's state space to determine whether certain states can be reached from a given initial state, while model checking involves checking a formal model of a system against a set of properties. Both techniques have their strengths and weaknesses and are used depending on the system's size and complexity.

REFERENCES

[1] C. Czepa, A. Amiri, E. Ntentos, and U. Zdun, "Modeling compliance specifications in linear temporal logic, event processing language and property specification patterns: a controlled experiment on understandability," *Softw. Syst. Model.*, 2019, doi: 10.1007/s10270-019-00721-4.

[2] E. Shishkin, "Debugging Smart Contract's Business Logic Using Symbolic Model Checking," *Program. Comput. Softw.*, 2019, doi: 10.1134/S0361768819080164.

[3] S. Carr, N. Jansen, R. Wimmer, A. Serban, B. Becker, and U. Topcu, "Counterexample-guided strategy improvement for POMDPs using recurrent neural networks," in *IJCAI International Joint Conference on Artificial Intelligence*, 2019. doi: 10.24963/ijcai.2019/768.

[4] B. K. Aichernig and M. Tappler, "Probabilistic black-box reachability checking (extended version)," *Form. Methods Syst. Des.*, 2019, doi: 10.1007/s10703-019-00333-0.

[5] J. Su and W. H. Chen, "Model-Based Fault Diagnosis System Verification Using Reachability Analysis," *IEEE Trans. Syst. Man, Cybern. Syst.*, 2019, doi: 10.1109/TSMC.2017.2710132.

[6] S. Haesaert, P. M. J. Van den Hof, and A. Abate, "Data-driven and model-based verification via Bayesian identification and reachability analysis," *Automatica*, 2017, doi: 10.1016/j.automatica.2017.01.037.

[7] E. M. Hahn, H. Hermanns, and L. Zhang, "Probabilistic reachability for parametric markov models," *Int. J. Softw. Tools Technol. Transf.*, 2011, doi: 10.1007/s10009-010-0146-x.

[8] N. Kobayashi, "Model checking higher-order programs," *J. ACM*, 2013, doi: 10.1145/2487241.2487246.

CHAPTER 23

THE ANALOG/DIGITAL INTERFACE

Prof. Sudhir Kumar Sharma, Associate Professor,
Department of Electronics & Communication Engineering, Jaipur National University, Jaipur, India,
Email Id-hodece_sadtm@jnujaipur.ac.in

Abstract:

An analog/digital interface refers to the methods and devices used to convert signals between the analog and digital domains. This interface is essential in many modern electronic systems that require the processing of both analog and digital signals. Analog signals are continuous waveforms that represent physical quantities such as sound, temperature, or light. In contrast, digital signals are discrete, consisting of a sequence of binary values (0 and 1) that represent information in a system. The conversion of analog signals to digital signals is known as analog-to-digital conversion (ADC), while the reverse process is called digital-to-analog conversion (DAC).

Keywords:

Analog Interface, Digital Interface, Digital Signal, Temperature, Physical Quantities.

INTRODUCTION

Analog-to-digital conversion (ADC) and digital-to-analog conversion (DAC) are critical elements of many modern electronic systems. These two types of conversion allow analog signals to be converted into digital data that can be processed and analyzed by digital circuits, and digital data to be converted back into analog signals for output to external devices. In this article, we will explore the basics of analog-to-digital and digital-to-analog conversion and their role in modern electronic systems [1].

ANALOG-TO-DIGITAL CONVERSION

Analog signals are continuous signals that vary over time and can take on any value within a certain range. Examples of analog signals include sound waves, voltage signals, and temperature readings. Digital circuits, on the other hand, operate on discrete values, such as binary values 0 and 1. To enable digital circuits to process and analyze analog signals, the analog signals must be converted to digital data. This process is called analog-to-digital conversion (ADC).

The basic process of ADC involves sampling, quantization, and encoding. Sampling involves taking discrete samples of an analog signal at regular intervals. The sampling rate determines how many samples are taken per second and is measured in hertz (Hz). The higher the sampling rate, the more accurately the original analog signal can be reconstructed from the digital data [2].

Quantization involves assigning a digital value to each sample of the analog signal. The number of bits used to represent each sample is called the resolution of the ADC. The higher the resolution, the more accurately the original analog signal can be reconstructed from the digital data. The most common resolutions are 8, 10, 12, or 16 bits[3], [4].

Encoding involves converting the quantized digital values into binary data that can be processed by digital circuits. The most common encoding method is binary encoding, which assigns a binary value to each digital value. For example, an 8-bit ADC with a resolution of 2.5 volts will assign a binary value of 00000000 to a voltage of 0 volts and a binary value of 11111111 to a voltage of 2.5 volts.

There are two main types of ADC: successive approximation ADC and delta-sigma ADC. Successive approximation ADC uses a binary search algorithm to quickly determine the digital value of each sample. Delta-sigma ADC uses a technique called oversampling, which involves taking multiple samples of the analog signal at a high sampling rate and then filtering and averaging the samples to improve the resolution of the digital data.

DIGITAL-TO-ANALOG CONVERSION

Digital-to-analog conversion (DAC) is the process of converting digital data into an analog signal. This process is essential for outputting digital data to external devices, such as speakers, displays, and motors that require an analog signal to operate. The basic process of DAC involves decoding, weighting, and summing. Decoding involves converting the binary data into digital values that can be used to generate an analog signal [5], [6]. The most common decoding method is binary decoding, which assigns a digital value to each binary value. For example, an 8-bit DAC with a range of 0 to 2.5 volts will assign a digital value of 0 to a binary value of 00000000 and a digital value of 255 to a binary value of 11111111.

Weighting involves assigning a weight to each digital value based on its position in the binary code. The weight of each bit is determined by its position in the binary code, with the most significant bit (MSB) having the highest weight and the least significant bit (LSB) having the lowest weight. For example, in an 8-bit DAC, the MSB would have a weight of 128 and the LSB would have a weight of 1.

Summing involves adding together the weighted digital values to generate the analog signal. The analog signal is generated by taking the sum of the weighted digital values and scaling the result to the desired range. The scaling factor is determined by the reference voltage, which is the maximum voltage that the DAC can output. For example, if the reference voltage is 2.5 volts and the sum of the weighted digital values is 1023, then the output voltage of the DAC will be 2.5 volts.

There are two main types of DAC: pulse-width modulation (PWM) DAC and current steering DAC. PWM DAC uses a technique called pulse-width modulation to generate an analog signal from a digital signal [7], [8]. The digital signal is converted into a series of pulses, with the width of each pulse proportional to the digital value. The pulses are then filtered to remove the high-frequency components, leaving only the low-frequency components that represent the analog signal. Current steering DAC uses a technique called current steering to generate an analog signal from a digital signal. The digital signal is converted into a series of currents, with the amplitude of each current proportional to the digital value. The currents are then combined and filtered to generate the analog signal.

DISCUSSION

INTERFACE CIRCUITS

An interface circuit is a circuit that connects an analog or digital device to a digital system. The interface circuit is responsible for converting the analog signal from the device into digital data that can be processed by the digital system, or for converting the digital data from the digital system into an analog signal that can be output to the device. Interface circuits can be used in a wide variety of applications, such as audio systems, sensors, and communication systems.

There are two main types of interface circuits: serial interface circuits and parallel interface circuits. Serial interface circuits transfer data one bit at a time, while parallel interface circuits transfer data multiple bits at a time. Serial interface circuits are typically used in applications where the data rate is low, while parallel interface circuits are typically used in applications where the data rate is high.

SERIAL INTERFACE CIRCUITS

Serial interface circuits transfer data one bit at a time, using a clock signal to synchronize the transfer. Serial interface circuits can be divided into two types: synchronous serial interface circuits and asynchronous serial interface circuits.

Synchronous serial interface circuits transfer data at a fixed rate, using a clock signal to synchronize the transfer. The clock signal is generated by the master device, which controls the transfer of data. The slave device receives the clock signal and uses it to synchronize the transfer of data. Synchronous serial interface circuits can transfer data in either direction, and can support multiple devices on a single bus.

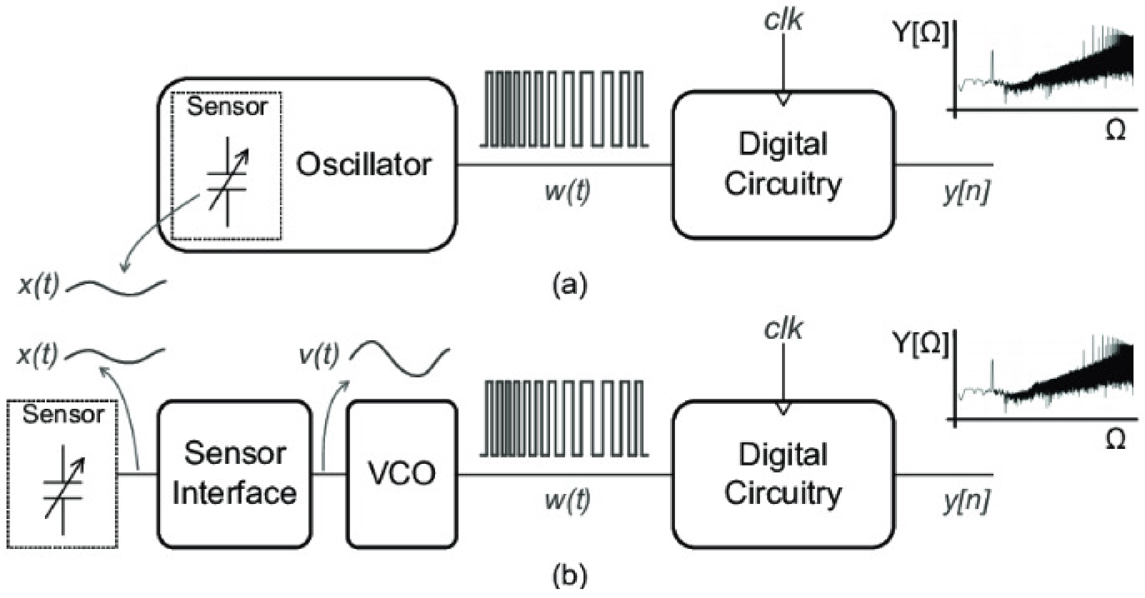


Figure 1: Illustrate the Oscillator-based analog-to-digital converter.

Asynchronous serial interface circuits transfer data at variable rates, without using a clock signal to synchronize the transfer. Instead, the transfer of data is controlled by start and stop bits, which indicate the beginning and end of each data transfer. Asynchronous serial interface circuits can only transfer data in one direction, and can typically only support two devices on a single bus.

PARALLEL INTERFACE CIRCUITS

Parallel interface circuits transfer data multiple bits at a time, using a set of data lines to transfer the data. Parallel interface circuits can be divided into two types: synchronous parallel interface circuits and asynchronous parallel interface circuits. Synchronous parallel interface circuits transfer data at a fixed rate, using a clock signal to synchronize the transfer. The clock signal is generated by the master device, which controls the transfer of data. The slave device receives the clock signal and uses it to synchronize the transfer of data. Synchronous parallel interface circuits can transfer data in either direction, and can support multiple devices on a single bus. In Figure 1 illustrate the oscillator-based analog-to-digital converter.

Analog-to-digital interfaces (ADCs) and digital-to-analog interfaces (DACs) are crucial components in many electronic devices that require conversion of analog signals into digital data or vice versa. These interfaces are commonly used in communication systems, control systems, measurement devices, and consumer electronics. We will explore the working principles of ADCs and DACs, their applications, and their limitations.

1. Analog-to-Digital Interfaces (ADCs)

An ADC is a device that converts analog signals into digital data that can be processed by digital systems. Analog signals are continuous signals that vary in amplitude or frequency over time. Examples of analog signals include sound, light, temperature, pressure, and voltage.

The digital data produced by an ADC is a stream of binary values that represent the amplitude of the analog signal at specific time intervals. These values are usually stored in memory or transmitted to other devices for further processing.

The process of analog-to-digital conversion involves two main steps: sampling and quantization.

Sampling is the process of measuring the amplitude of an analog signal at specific time intervals. The resulting samples are discrete values that approximate the continuous analog signal. The sampling rate, or frequency, determines the number of samples taken per second and is measured in Hertz (Hz).

To ensure accurate representation of the original analog signal, the sampling rate must be at least twice the highest frequency of the signal. This is known as the Nyquist-Shannon sampling theorem, and it guarantees that the analog signal can be reconstructed from the digital samples without loss of information.

Quantization is the process of mapping each sample to a specific digital value. The number of possible digital values is determined by the bit depth of the ADC, which represents the number of bits used to represent each sample. The more bits used, the more accurate the digital representation of the analog signal.

However, increasing the bit depth also increases the complexity and cost of the ADC. Therefore, the bit depth is usually determined by the required accuracy of the application.

Types of ADCs

There are several types of ADCs, including:

a) **Flash ADC:** This type of ADC uses a bank of comparators to quickly compare the input voltage to a set of reference voltages. The output of the comparators is then encoded into a binary value.

b) **Successive Approximation ADC:** This type of ADC uses a binary search algorithm to iteratively approximate the input voltage. The algorithm starts with the most significant bit and continues until the desired accuracy is achieved.

c) **Delta-Sigma ADC:** This type of ADC uses a modulator to convert the analog signal into a digital stream of 1s and 0s. The output of the modulator is then filtered to remove noise and obtain the final digital output.

2. Digital-to-Analog Interfaces (DACs)

A DAC is a device that converts digital data into analog signals. Digital data is a series of binary values that represent the amplitude of a signal at specific time intervals. The process of digital-to-analog conversion involves two main steps: reconstruction and smoothing.

Reconstruction

Reconstruction is the process of converting the discrete digital values into a continuous analog signal. This is achieved using a reconstruction filter, which interpolates between the digital values to generate a continuous signal. The quality of the reconstructed signal depends on the sampling rate and the order of the reconstruction filter. Higher sampling rates and filter orders result in a more accurate reconstruction of the original analog signal.

Smoothing

Smoothing is the process of removing any high-frequency noise or distortion from the reconstructed signal. This is achieved using a low-pass filter, which removes frequencies above a certain cutoff frequency. The cutoff frequency is determined by the bandwidth of the original analog signal and the required accuracy of the reconstructed signal. A higher cutoff frequency results in more distortion in the reconstructed signal, while a lower cutoff frequency results in a smoother but less accurate signal.

Types of DACs

There are several types of DACs, including:

a) **Binary-weighted DAC:** This type of DAC uses a network of resistors to generate reference voltages that are weighted according to the binary value of the input digital signal. The weighted voltages are then combined to generate the final analog output.

b) **R-2R ladder DAC:** This type of DAC uses a ladder network of resistors to generate the reference voltages. The ladder network consists of two sets of resistors, with one set having a value twice that of the other set. The output voltage is generated by summing the voltages across the ladder network.

c) **Delta-Sigma DAC:** This type of DAC uses a modulator to convert the digital input signal into a stream of 1s and 0s. The output of the modulator is then filtered to remove noise and obtain the final analog output.

3. Applications of ADCs and DACs

ADCs and DACs are used in a wide range of applications, including:

Communication systems

ADCs and DACs are used in communication systems to convert analog signals into digital data and vice versa. Examples of communication systems that use ADCs and DACs include modems, digital telephony, and wireless communication systems.

In these systems, ADCs are used to sample analog signals and convert them into digital data that can be transmitted over a communication channel. DACs are used at the receiving end to reconstruct the original analog signal from the received digital data.

Control systems

ADCs and DACs are used in control systems to measure and control physical variables such as temperature, pressure, and position. In these systems, ADCs are used to measure the analog signals, and DACs are used to generate control signals based on the digital data.

For example, in a temperature control system, an ADC is used to measure the temperature, and a DAC is used to generate a control signal that adjusts the heating or cooling system to maintain the desired temperature.

Measurement devices

ADCs are used in measurement devices to sample analog signals and convert them into digital data that can be analyzed and processed. Examples of measurement devices that use ADCs include oscilloscopes, spectrum analyzers, and data loggers.

In these devices, the accuracy and resolution of the ADC are crucial for obtaining accurate and reliable measurement data. DACs are also used in some measurement devices to generate control signals that adjust the test conditions or simulate test signals.

Consumer electronics

ADCs and DACs are used in many consumer electronics devices, such as digital cameras, MP3 players, and smartphones. In these devices, ADCs are used to convert analog signals, such as sound or light, into digital data that can be processed by the device.

DACs are used to convert digital data, such as music or video files, into analog signals that can be played back through speakers or displays.

4. Limitations of ADCs and DACs

ADCs and DACs have several limitations that can affect their performance and accuracy.

Noise and distortion

Noise and distortion can affect the accuracy and reliability of ADCs and DACs. Noise is an unwanted signal that is added to the signal being measured or generated, while distortion is a change in the shape or frequency of the signal.

The sources of noise and distortion in ADCs and DACs include the electronic components, the analog signal being measured or generated, and the digital processing circuits. To minimize the effects of noise and distortion, ADCs and DACs are designed with low noise and distortion specifications and are often shielded.

HANDSHAKING SIGNALS

Handshaking signals are signals used in parallel interface circuits to synchronize the transfer of data between two devices. Handshaking signals include:

1. **Data Valid:** This signal indicates that valid data is present on the data lines.
2. **Ready:** This signal indicates that the receiving device is ready to accept data.
3. **Acknowledge:** This signal indicates that the sending device has received the data.
4. **Error:** This signal indicates that an error has occurred during the transfer of data.

Handshaking signals are used to ensure that data is transferred reliably between the two devices. The sending device waits for the ready signal before sending the data, and the receiving device sends an acknowledge signal to indicate that the data has been received. If an error occurs during the transfer of data, the error signal is used to alert both devices that an error has occurred.

I/O PORTS

Input/output (I/O) ports are used in digital systems to connect external devices, such as sensors, switches, and displays, to the system. I/O ports can be divided into two types: input ports and output ports. Input ports are used to receive data from external devices. The data is read from the input port and processed by the digital system. Input ports can be used to connect a wide variety of external devices, such as sensors, switches, and keyboards.

Output ports are used to send data to external devices. The data is written to the output port and output to the external device. Output ports can be used to connect a wide variety of external devices, such as displays, motors, and LEDs. I/O ports are typically implemented using parallel interface circuits. Each port consists of a set of data lines, as well as handshaking signals to ensure reliable data transfer.

CONCLUSION

ADCs and DACs play a critical role in modern electronic systems, allowing digital devices to interface with the analog world. ADCs are used to sample and convert analog signals into digital data, while DACs are used to convert digital data back into analog signals. The accuracy and performance of ADCs and DACs are influenced by factors such as resolution, sampling rate, and noise and distortion, and their limitations need to be considered when designing electronic systems. ADCs and DACs are used in a wide range of applications, including communication systems, control systems, measurement devices, and consumer electronics. Their ability to convert between the analog and digital domains enables the processing and analysis of signals, making them indispensable for many electronic systems. ADCs and DACs have revolutionized the way we interact with the analog world, enabling us to create sophisticated electronic systems that can process and analyze complex signals. As technology continues to advance, the demand for high-performance ADCs and DACs will only increase, driving innovation and development in this field.

REFERENCES

[1] T. Cooklev, R. Normoyle, and D. Clendenen, “The VITA 49 analog RF-digital interface,” *IEEE Circuits Syst. Mag.*, 2012, doi: 10.1109/MCAS.2012.2221520.

[2] C. J. Gonzalez *et al.*, “Reducing Soft Error Rate of SoCs Analog-to-Digital Interfaces with Design Diversity Redundancy,” *IEEE Trans. Nucl. Sci.*, 2020, doi: 10.1109/TNS.2019.2952775.

[3] I. Haghighi, N. Mehdipour, E. Bartocci, and C. Belta, “Control from Signal Temporal Logic Specifications with Smooth Cumulative Quantitative Semantics,” in *Proceedings of the IEEE Conference on Decision and Control*, 2019. doi: 10.1109/CDC40024.2019.9029429.

[4] A. Hussain, J. K. Singh, A. R. Senthil Kumar, and K. R. Harne, “Rainfall-runoff modeling of sutlej river basin (India) using soft computing techniques,” *Int. J. Agric. Environ. Inf. Syst.*, 2019, doi: 10.4018/IJAEIS.2019040101.

[5] T. S. Nouidui, M. Wetter, Z. Li, X. Pang, P. Bhattacharya, and P. Haves, “Bacnet and analog/digital interfaces of the building controls virtual testBed,” in *Proceedings of Building Simulation 2011: 12th Conference of International Building Performance Simulation Association*, 2011.

[6] N. van Bavel, K. S. Albright, and X. M. Gong, “An Analog/Digital Interface for Cellular Telephony,” *IEEE J. Solid-State Circuits*, 1995, doi: 10.1109/4.364433.

[7] A. R. Gaiduk, N. N. Prokopenko, and A. V. Bugakova, “Accuracy Increase of Discrete Sensors with Time Delay,” *IEEE Sens. J.*, 2020, doi: 10.1109/JSEN.2020.2996079.

[8] H. Taoka, I. Iyoda, H. Noguchi, N. Sato, T. Nakazawa, and A. Yamazaki, “Real-Time Digital Simulator with Digital/Analog Conversion Interface for Testing Power Instruments,” *IEEE Trans. Power Syst.*, 1994, doi: 10.1109/59.317628.