Discrete Event Systems

Introduction

Why should you care?

Being based on natural phenomena, science is often explained by continuous variables.

Discrete Event Systems

\[ F = G \frac{m_1 m_2}{r^2} \]

Mechanics

Gravitation

Electrodynamic
Being based on natural phenomena, Science is often explained by continuous variables.

Many complex systems are not continuous...

\[ F = G \frac{m_1 m_2}{r^2} \]

solved by differential equations

Somewhere inside Google datacenters

computer systems

NYC subway system

transportation systems
Those systems are determined by discrete events

- Customers requests
- Telephone calls
- Train arrivals
- Incoming data
- Equipment failures

In this course, you'll learn how to design, model, analyze, test, and optimize discretized event systems.

Some examples:

- Automata & Petri nets
- Average-, worst-case viewpoint
- Out of a specification
- Proof system properties
- Minimize the system size
There will be 3 professors in the course

Part I
- Laurent Vanbever
  - Automatas

Part II
- Roger Wattenhofer
  - Stochastic process

Part III
- Lothar Thiele
  - Specification model

Week 1-5
- Laurent Vanbever
- Roger Wattenhofer
- Lothar Thiele
- Automatas
- Stochastic process
- Specification model

Week 6-10
- Laurent Vanbever
- Roger Wattenhofer
- Lothar Thiele

Week 11-13
- Laurent Vanbever
- Roger Wattenhofer
- Lothar Thiele
- Automatas
- Stochastic process
- Specification model

Course organization

- Lectures: Thursday 1pm-3pm
  @ETZ 9

- Exercices: Thursday 3pm-5pm
  @ETZ 9

- Materials: http://www.disco.ethz.ch/lectures/des/

Automata & languages
A primer on the Theory of Computation

Laurent Vanbever
www.vanbever.eu

ETH Zürich (D-ITET)
September, 17 2015
The imitation game (2014)
Benedict Cumberbatch

Alan Turing (1912-1954)

Brief CV
created computer science as we know it
invented a universal computer model, the Turing machine
broke German cyphers, most notably Enigma
invented the Turing Test to distinguish human from machine

Can a computer compute anything?

invented a universal computer model, the Turing machine
studied the fundamental limitations of computers
Given an arbitrary program $P$ & an input $I$

Decide whether $P$ will
- halt, eventually
- loop, forever

when run on $I$

This problem cannot be solved by a computer (no matter its power)

Many other problems were shown to be “uncomputable”

Given a set of dominos:

Post-correspondance problem
Emil Post, 1946

Given a set of dominos:

Post-correspondance problem
Emil Post, 1946

Make a list of them s.t.
the top string equals
the bottom one, e.g.

(repetitions are allowed)

In this part of the course, we'll learn about what is “computable” and “not”.

We’ll study three models of computation, from the least powerful to the most expressive:

1. **Part 1**: Regular language
   - Vendor machines
2. **Part 2**: Context-free language
   - Programming languages
3. **Part 3**: Turing machine
   - Computer

Automata & languages
A primer on the Theory of Computation
## Part I: Regular Languages

<table>
<thead>
<tr>
<th>Finite Automata</th>
<th>1.</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thu Sept 17</td>
<td>2.</td>
<td>Definition</td>
</tr>
<tr>
<td></td>
<td>3.</td>
<td>Design</td>
</tr>
<tr>
<td></td>
<td>4.</td>
<td>Regular operations</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• closure</td>
</tr>
<tr>
<td></td>
<td></td>
<td>• union <em>et al.</em></td>
</tr>
</tbody>
</table>

### The Coke Vending Machine

- Vending machine dispenses soda for $0.45 a pop.
- Accepts only dimes ($0.10) and quarters ($0.25).
- Eats your money if you don’t have correct change.
- You’re told to “implement” this functionality.

#### Vending Machine Java Code

```java
Soda vend()
{
    int total = 0, coin;
    while (total != 45)
    {
        receive(coin);
        if ((coin==10 && total==40) || (coin==25 && total>=25))
            reject(coin);
        else
            total += coin;
    }
    return new Soda();
}
```

### Why this was overkill...

- Vending machines have been around long before computers.
  - Or Java, for that matter.
- Don’t really need int’s.
  - Each int introduces $2^{32}$ possibilities.
- Don’t need to know how to add integers to model vending machine
  - total += coin.
- Java grammar, if-then-else, etc. complicate the essence.
Vending Machine “Logics”

Why was this simpler than Java Code?

• Only needed two coin types “D” and “Q”
  – symbols/letters in alphabet
• Only needed 7 possible current total amounts
  – states/nodes/vertices
• Much cleaner and more aesthetically pleasing than Java lingo
• Next: generalize and abstract...

Alphabets and Strings

• Definitions:
  - An alphabet $\Sigma$ is a set of symbols (characters, letters).
  - A string (or word) over $\Sigma$ is a sequence of symbols.
    - The empty string is the string containing no symbols at all, and is denoted by $\varepsilon$.
    - The length of the string is the number of symbols, e.g. $|\varepsilon| = 0$.

Questions:
1) What is $\Sigma$?
2) What are some good or bad strings?
3) What does $\varepsilon$ signify here?
Finite Automaton Example

Formal Definition of a Finite Automaton

A finite automaton (FA) is a 5-tuple \((Q, \Sigma, \delta, q_0, F)\), where

- \(Q\) is a finite set called the states
- \(\Sigma\) is a finite set called the alphabet
- \(\delta: Q \times \Sigma \rightarrow Q\) is the transformation function
- \(q_0 \in Q\) is the start state
- \(F \subseteq Q\) is the set of accept states (a.k.a. final states).

Accept States

- How does an FA operate on strings?

Imagine an auxiliary tape containing the string.

The FA reads the tape from left to right with each new character causing the FA to go into another state.

When the string is completely read, the string is accepted depending on whether the FA's final state was an accept state.

Further explaining is needed for understanding how FA's interact with their input.
Accept States

• How does an FA operate on strings?
  Imagine an auxiliary tape containing the string.
  The FA reads the tape from left to right with each new character
  causing the FA to go into another state.
  When the string is completely read, the string is accepted
  depending on whether the FA’s final state was an accept state.

• Definition: A string $u$ is accepted by an automaton $M$ if and only if
  the path starting at $q_0$, which is labeled by $u$, ends in an accept state.

Language

• Definition:
  The language accepted by a finite automaton $M$ is the set of all strings
  which are accepted by $M$. The language is denoted by $L(M)$.
  We also say that $M$ recognizes $L(M)$, or that $M$ accepts $L(M)$.

• Think of all the possible ways of getting from the start to any accept state.

• We will eventually see that not all languages can be described as the
  accepted language of some FA.

• A language $L$ is called a regular language if there exists a FA $M$ that
  recognizes the language $L$.
Designing Finite Automata

• This is essentially a creative process...

• "You are the automaton" method
  – Given a language (for which we must design an automaton).
  – Pretending to be automaton, you receive an input string.
  – You get to see the symbols in the string one by one.
  – After each symbol you must determine whether string seen so far is part of the language.
  – Like an automaton, you don’t see the end of the string, so you must always be ready to answer right away.

• Main point: What is crucial, what defines the language?!

Find the automata for...

1) \( \Sigma = \{0,1\} \)
   Language consists of all strings with odd number of ones.

2) \( \Sigma = \{0,1\} \)
   Language consists of all strings with substring "001",
   for example 100101, but not 1010110101.

More examples in the book and in the exercises...

Definition of Regular Language

• Recall the definition of a regular language:
  A language \( L \) is called a regular language if there exists a FA \( M \) that recognizes the language \( L \).

• We would like to understand what types of languages are regular. Languages of this type are amenable to super-fast recognition.

Definition of Regular Language

• Recall the definition of a regular language:
  A language \( L \) is called a regular language if there exists a FA \( M \) that recognizes the language \( L \).

• We would like to understand what types of languages are regular. Languages of this type are amenable to super-fast recognition.

• Are the following languages regular?
  – Unary prime numbers: \{ 11, 111, 1111, 11111, 111111, 1111111, ... \} = \{ 1^p | p is a prime number \}
  – Palindromic bit strings: \{ 0, 00, 100, 111, 1101, 1111, ... \}
Finite Languages

- All the previous examples had the following property in common: *infinite cardinality*
- Before looking at infinite languages, we should look at finite languages.

Languages of Cardinality 1

- Answer: Yes.

- Question: Huh? What’s wrong with this automaton?!? What if the automation is in state $q_1$ and reads a “b”?

Languages of Cardinality 1

- Answer: Yes.

- Question: Is the singleton language containing one string regular? For example, is the language {banana} regular?
Languages of Cardinality 1

- Answer: Yes.
- Question: Huh? What’s wrong with this automaton?!? What if the automation is in state q₁ and reads a “b”?
  - Answer: This a first example of a nondeterministic FA. The difference between a deterministic FA (DFA) and a nondeterministic FA (NFA) is that every DFA state has one exiting transition arrow for each symbol of the alphabet.

Arbitrary Finite Number of Finite Strings

- Theorem: All finite languages are regular.
  - Proof:
    
    One can always construct a tree whose leaves are word-ending. Make word endings into accept states, add a fail sink-state and add links to the fail state to finish the construction.

    Since there’s only a finite number of finite strings, the automaton is finite.
## Arbitrary Finite Number of Finite Strings

- **Theorem:** All finite languages are regular.
- **Proof:**
  One can always construct a tree whose leaves are word-ending. Make word endings into accept states, add a fail sink-state and add links to the fail state to finish the construction.

Since there’s only a finite number of finite strings, the automaton is finite.

- **Example for \{banana, nab, ban, babba\}:

```
  b  a  n  b  a
  a  n  b
  a  n
```

---

## Regular Operations

- **You may have come across the regular operations when doing advanced searches utilizing programs such as emacs, egrep, perl, python, etc.**
- **There are four basic operations we will work with:**
  - Union
  - Concatenation
  - Kleene-Star
  - Kleene-Plus (which can be defined using the other three)

---

### Regular Operations – Summarizing Table

<table>
<thead>
<tr>
<th>Operation</th>
<th>Symbol</th>
<th>UNIX version</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Union</td>
<td>∪</td>
<td></td>
<td>Match one of the patterns</td>
</tr>
<tr>
<td>Concatenation</td>
<td>•</td>
<td>implicit in UNIX</td>
<td>Match patterns in sequence</td>
</tr>
<tr>
<td>Kleene-star</td>
<td>*</td>
<td>*</td>
<td>Match pattern 0 or more times</td>
</tr>
<tr>
<td>Kleene-plus</td>
<td>+</td>
<td>+</td>
<td>Match pattern 1 or more times</td>
</tr>
</tbody>
</table>
Regular operations: Union

• In UNIX, to search for all lines containing vowels in a text one could use
the command
  ```bash
  egrep -i 'a|e|i|o|u'
  ```
  Here the pattern "vowel" is matched by any line containing a vowel.
  A good way to define a pattern is as a set of strings, i.e. a language.
  The language for a given pattern is the set of all strings satisfying the
  predicate of the pattern.

• In UNIX, to search for all lines containing vowels in a text one could use
the command
  ```bash
  egrep -i 'a|e|i|o|u'
  ```
  Here the pattern "vowel" is matched by any line containing a vowel.
  A good way to define a pattern is as a set of strings, i.e. a language.
  The language for a given pattern is the set of all strings satisfying the
  predicate of the pattern.

• In UNIX, a pattern is implicitly assumed to occur as a substring of the
matched strings. In our course, however, a pattern needs to specify
the whole string, not just a substring.

• Computability: Union is exactly what we expect.
  If you have patterns
  \[ A = \{aardvark\}, B = \{bobcat\}, C = \{chimpanzee\} \]
  the union of these is
  \[ A \cup B \cup C = \{aardvark, bobcat, chimpanzee\} \]

Regular operations: Concatenation

• To search for all consecutive double occurrences of vowels, use:
  ```bash
  egrep -i 'a|e|i|o|u a|e|i|o|u'
  ```
  Here the pattern "vowel" has been repeated. Parentheses have been
  introduced to specify where exactly in the pattern the concatenation is
  occurring.
Regular operations: Concatenation

- To search for all consecutive double occurrences of vowels, use:
  - `egrep -i `(a|e|i|o|u)(a|e|i|o|u)'
  - Here the pattern "vowel" has been repeated. Parentheses have been introduced to specify where exactly in the pattern the concatenation is occurring.

- Computability: Consider the previous result:
  \[ L = \{ \text{aardvark, bobcat, chimpanzee} \} \]
  
  When we concatenate \( L \) with itself we obtain:
  \[ L \times L = \{ \text{aardvark, bobcat, chimpanzee} \} \times \{ \text{aardvark, bobcat, chimpanzee} \} = \]
  \{ aardvarkaardvark, aardvarkbobcat, aardvarkchimpanzee, bobcatbobcat, babababa, banana, babana, naba, nana, bababa, babababa, ... \}

Regular operations: Kleene-*

- We continue the UNIX example: now search for lines consisting purely of vowels (including the empty line):
  - `egrep -i `^\(a|e|i|o|u\)*$'`
  - Note: ^ and $ are special symbols in UNIX regular expressions which respectively anchor the pattern at the beginning and end of a line. The trick above can be used to convert any Computability regular expression into an equivalent UNIX form.

- Computability: Suppose we have a language \( B = \{ \text{ba, na} \} \).
  Question: What is the language \( B^* \)?

  Answer: \( B^* = \{ \text{ba, na} \}^* = \{ \varepsilon, \text{ba, na, bana, naba, nana, bababa, babababa, ... } \} \)
Regular operations: Kleene-+

- Kleene-+ is just like Kleene-* except that the pattern is forced to occur at least once.
- UNIX: search for lines consisting purely of vowels (not including the empty line):
  ```
  egrep -i '^([aeiou]+)$'
  ```

- Computability:
  Suppose we have a language \( \mathcal{B} = \{ba, na\} \).
  What is \( \mathcal{B}^+ \) and how does it defer from \( \mathcal{B}^* \)?
  \[\mathcal{B}^+ = \{ba, na, baba, bana, naba, nana, bababa, babana, bana, nanaba, nanana, babababa, bababana, ... \}\]
  The only difference is the absence of \( \epsilon \).

Closure of Regular Languages

- When applying regular operations to regular languages, regular languages result. That is, regular languages are closed under the operations of union, concatenation, and Kleene-*.
- Goal: Show that regular languages are closed under regular operations.
  In particular, given regular languages \( L_1 \) and \( L_2 \), show:
  1. \( L_1 \cup L_2 \) is regular,
  2. \( L_1 \cdot L_2 \) is regular,
  3. \( L_1^* \) is regular.

- No.'s 2 and 3 are deferred until we learn about NFA's.
- However, No. 1 can be achieved immediately.
Union Example

- Problem: Draw the FA for
  \[ L = \{ x \in \{0,1\}^* \mid |x|\text{even or } x \text{ ends with 11} \} \]

Let’s start by drawing \( M_1 \) and \( M_2 \), the automaton recognizing \( L_1 \) and \( L_2 \)

- \( L_1 = \{ x \in \{0,1\}^* \mid x \text{ has even length} \} \)
- \( L_2 = \{ x \in \{0,1\}^* \mid x \text{ ends with 11} \} \)

Cartesian Product Construction

- We want to construct a finite automaton \( M \) that recognizes any strings belonging to \( L_1 \) or \( L_2 \).
- Idea: Build \( M \) such that it simulates both \( M_1 \) and \( M_2 \) simultaneously and accept if either of the automatons accepts.

Definition: The Cartesian product of two sets \( A \) and \( B \), denoted by \( A \times B \), is the set of all ordered pairs \( (a,b) \) where \( a \in A \) and \( b \in B \).
Formal Definition

- Given two automata
  \( M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1) \) and \( M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2) \)

- Define the unioner of \( M_1 \) and \( M_2 \) by:
  \[
  M \cup = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_1, q_2), F_\cup)
  \]
  - where the accept state \( (q_1, q_2) \) is the combined start state of both automata
  - where \( F_\cup \) is the set of ordered pairs in \( Q_1 \times Q_2 \) with at least one state an accept state. That is:
  \[
  F_\cup = F_1 \times Q_2 \cup Q_1 \times F_2
  \]
  - where the transition function \( \delta \) is defined as
  \[
  \delta((q_1, q_2), j) = (\delta_1(q_1, j), \delta_2(q_2, j)) = \delta_1 \times \delta_2
  \]

Other constructions: Intersector

- Other constructions are possible, for example an intersector:

- Accept only when both ending states are accept states. So the only difference is in the set of accept states. Formally the intersector of \( M_1 \) and \( M_2 \) is given by
  \[
  M_\cap = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_0, q_0), F_\cap), \text{ where } F_\cap = F_1 \times F_2.
  \]
Other constructions: Difference

- The difference of two sets is defined by $A - B = \{x \in A \mid x \not\in B\}$
- In other words, accept when first automaton accepts and second does not.

$$M_\Delta = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_0,1,q_0,2), F_\Delta),$$
where $F_\Delta = F_1 \times Q_2 - Q_1 \times F_2$

Other constructions: Symmetric difference

- The symmetric difference of two sets $A, B$ is $A^\Delta B = A^\cap B - A^\cap \overline{B}$
- Accept when exactly one automaton accepts:

$$M_\oplus = (Q_1 \times Q_2, \Sigma, \delta_1 \times \delta_2, (q_0,1,q_0,2), F_\oplus),$$
where $F_\oplus = F_1 \cup F_2 - F_1 \cap F_2$
Complement

- How about the complement? The complement is only defined with respect to some universe.

- Given the alphabet \( \Sigma \), the default universe is just the set of all possible strings \( \Sigma^* \). Therefore, given a language \( L \) over \( \Sigma \), i.e. \( L \subseteq \Sigma^* \), the complement of \( L \) is \( \Sigma^* - L \).

- Note: Since we know how to compute set difference, and we know how to construct the automaton for \( \Sigma^* \) we can construct the automaton for \( L \).

- Question: Is there a simpler construction for \( L \)?

- Answer: Just switch accept-states with non-accept states.
Boolean-Closure Summary

- We have shown constructively that regular languages are closed under boolean operations. I.e., given regular languages $L_1$ and $L_2$ we saw that:
  1. $L_1 \cup L_2$ is regular,
  2. $L_1 \cap L_2$ is regular,
  3. $L_1 - L_2$ is regular,
  4. $L_1 \oplus L_2$ is regular,
  5. $L_1^c$ is regular.

- No. 2 to 4 also happen to be regular operations. We still need to show that regular languages are closed under concatenation and Kleene-*.

- Tough question: Can’t we do a similar trick for concatenation?