



Elements of System Dynamics

Antoine B. Rauzy

PART 2. THE Σ^{TM} LANGUAGE

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

Key notions:

- Grammars, EBNF
- Model-as-Script, Paths
- State Variables, Activity Variables, Instruction Variables
- Constants, Parameters, Ports, Observers
- Instructions, Expressions
- Random deviates
- External Data Sets

Learning Objectives

This lecture dives deeper into the Σ^{TM} language and introduces quite a few features.

You do not need to memorize all of them. But you certainly need to know that they exist, so that you can refer to the Σ^{TM} reference manual (available from the help of the WorldLabTM Workshop).

Agenda

Section 1. Formal Grammars

Section 2. The Model-as-Script Principle

Section 3. Variables

Section 4. Constants, Parameters, Ports

Section 5. Observers

Section 6. Instructions

Section 7. Expressions

Section 8. Random Deviates

Section 9. External Data Sets

Section 10. Comments

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 1. FORMAL GRAMMARS

Textual Formats

Models are not free texts. They are written according to **strict rules** making it possible to decide without ambiguity whether a text is a **well-formed model** or not.

```
system SoftwareDevelopment
  system Company
    system Alice
      Mode mode (init = STANDBY);
    end
    system Bob
      Mode mode (init = STANDBY);
    end
  end
  system Software
    int specifiedFeatures (init = 0);
    int codedFeatures (init = 0);
    int testedFeatures (init = 0);
  end
end
```

Languages

Let A be a denumerable set of symbols, called the **alphabet**.

A^* denotes the set of all finite sequences of symbols of A .

A **language** built over A is a subset of A^* .

- The alphabet of Σ^{TM} is made of:
 - The keywords `system`, `end`, `activity`...
 - The terminal symbols `'`, `,`, `'`, `;`, `'`, `:`, `'`...
 - The identifiers for systems, variables, activities...
- The language consists of all sequences of these symbols that are valid Σ^{TM} models.

Formal Grammars

Programming and modeling languages are in general described by means formal grammars. A **formal grammar** describes which strings from an alphabet of a formal language are valid. A grammar does not describe the meaning, only their form. A formal grammar is defined as a set of **production rules** for such strings in a formal language.

Consider the grammar made of the two production rules:

1. $S \rightarrow aSb$
2. $S \rightarrow ab$

The grammar defines the language $ab, aabb, aaabbb\dots$ i.e. $a^n b^n$

Reading a text and verifying that it is valid according to a certain grammar is called **parsing** this text (according to this grammar).

If all left-hand sides of rules of a grammar are made of a unique non-terminal symbol, the grammar is said **context-free** (and **context-dependent** otherwise). A context-free language is a language that can be parsed with a context-free grammar.

Context-free languages are much easier to parse than context-dependent ones.

Most of the programming and modeling languages, including Σ^{TM} , are context-free.

Extended Backus-Naur Form

The **Extended Backus-Naur Form (EBNF)** is a widely used way to write formal grammars.

Expression	Meaning
$S ::= E$	The non-terminal symbol S is defined by the expression E
$E F$	E followed F
$E \mid F$	E or F
E^*	Any number of E
E^+	Any positive number of E
$E?$	0 or 1 E
(E)	Grouping
$'...'$	Terminal symbol

EBNF Grammar for System Declarations

```
SystemDeclaration ::=
    'system' Path ObjectDeclaration* 'end'

Path :=
    Identifier ( '.' Identifier ) *
Identifier ::= [a-zA-Z_][a-zA-Z0-9-_]*

ObjectDeclaration ::=
    ConstantDeclaration | ParameterDeclaration
    | PortDeclaration
    | VariableDeclaration
    | ObserverDeclaration
    | SystemDeclaration
    | InstanceDeclaration
    | ActivityDeclaration
    | ExtendsDirective | ClonesDirective
    | ActualizesDirective
    | AttributeDeclaration
```

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 2. THE MODEL-AS-SCRIPT PRINCIPLE

Model-as-Script Principle

Σ^{TM} is an **object-oriented language** of the **S2ML+X family** (more on that in Lecture 7).

One of the key feature of this family of languages is that the **model as assessed** by assessment tools is obtained through a **compilation process** from the **model as designed** by the analyst.

In other words, models as designed can be seen as scripts whose executions produce the models as assessed. This is called the **model-as-script principle**.

The compilation process is **automated** and **preserves the semantics** of the original model. This makes it possible to introduce powerful features in the languages, such as object- and prototype-oriented constructs, and to “eliminate” them through the compilation process.

This has a drawback however: because the model as designed is seen as a script, the **order** in which its constituents are declared **matters**. Models are not purely declarative.

Forward Declarations (1)

When the model is large, it may be tedious and error prone to declare it in one block.

```
domain Mode {STANDBY, WORKING}

system SoftwareDevelopment
  system Company
    system Alice
      Mode mode (init = STANDBY);
    end
    system Bob
      Mode mode (init = STANDBY);
    end
  end
  system Software
    int specifiedFeatures (init = 0);
    int codedFeatures (init = 0);
    int testedFeatures (init = 0);
  end
end
```

Forward Declarations (2)

```
domain Mode {STANDBY, WORKING}
```

```
system SoftwareDevelopment
```

```
  system Company
```

```
    system Alice ... end
```

```
    system Bob ... end
```

```
  end
```

```
  system Software ... end
```

```
end
```

```
system SoftwareDevelopment.Company.Alice
```

```
  Mode mode(init = STANDBY);
```

```
end
```

```
system SoftwareDevelopment.Company.Bob
```

```
  Mode mode(init = STANDBY);
```

```
end
```

```
system SoftwareDevelopment.Software
```

```
  int specifiedFeatures(init = 0);
```

```
  int codedFeatures(init = 0);
```

```
  int testedFeatures(init = 0);
```

```
end
```

Comment

Dot notation

Forward Declarations (3)

We already applied this principle for activities.

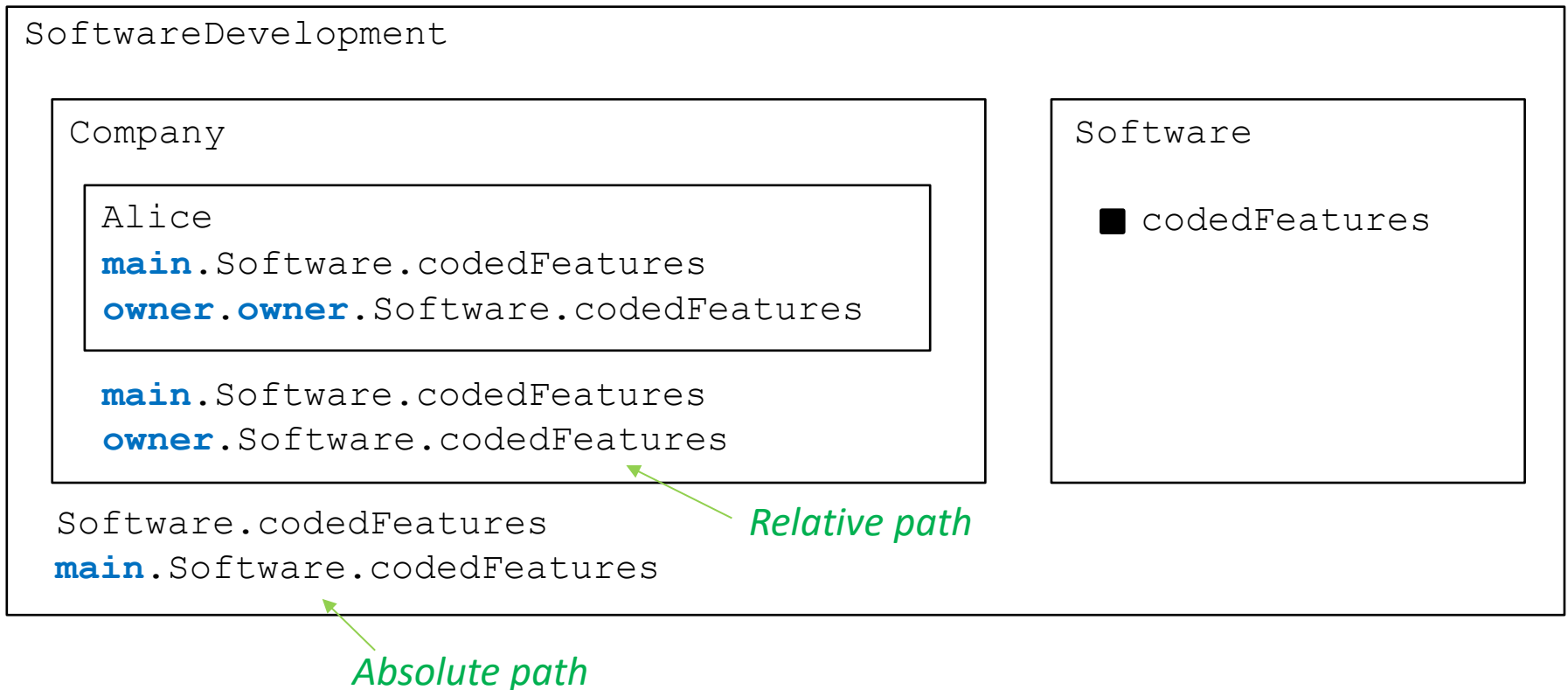
```
activity SoftwareDevelopment.Company.Alice.Specify
  trigger:
    return mode==STANDBY and main.Software.specifiedFeatures==0
    and main.Software.testedFeatures==0;
  start:
    mode = WORKING;
  completion: {
    main.Software.specifiedFeatures = 12;
    mode = STANDBY;
  }
  duration:
    return 3;
end
```

SoftwareDevelopment



Paths

Absolute and **relative paths** make it possible to access from any block to an element located in any other block of the hierarchy.



Redeclaring Systems

Redeclaring a system twice merges the contents of the two declarations.

```
system SoftwareDevelopment.Software  
  int specifiedFeatures(init = 0);  
  int codedFeatures(init = 0);  
end  
  
system SoftwareDevelopment.Software  
  int testedFeatures(init = 0);  
end
```

is equivalent to:

```
system SoftwareDevelopment.Software  
  int specifiedFeatures(init = 0);  
  int codedFeatures(init = 0);  
  int testedFeatures(init = 0);  
end
```

Redeclaring Base Objects

Redeclarations of base objects is also allowed. However,

1. The redeclaration must be compatible with the original declaration.
2. The redeclaration changes the original declaration.

```
system SoftwareDevelopment
  system Company
    system Alice
      parameter int codingRate = 2;
      Mode mode(init = STANDBY);
    end
    ...
  end
  ...
end
...
parameter int SoftwareDevelopment.Company.Alice.codingRate = 3;
```

Explained in next section

The value of the parameter `codingRate` is eventually equal to 3

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 3. VARIABLES

EBNF Grammar for Variable Declarations

Discussed in Lecture 5



```
VariableDeclaration ::= 'virtual'? Type Path AttributeList ';'

Type ::= BasicType | Domain
BasicType ::= 'bool' | 'int' | 'float' | 'id' | 'str'

DomainDeclaration ::= 'domain' Identifier SymbolicConstantSet
SymbolicConstantSet ::= '{' SymbolicConstants '}'
SymbolicConstants ::= SymbolicConstant (',' SymbolicConstant)*

Domain ::= Identifier
SymbolicConstant ::= Identifier

AttributeList ::= '(' Attributes ')'
Attributes ::= Attribute (',' Attribute)*
Attribute ::= Identifier '=' Expression
```

Types and Domains

Basic types are the following.

- **bool** for Boolean values;
- **int** for integers;
- **float** for floating-point numbers;
- **id** for identifiers;
- **str** for strings

A **user defined domain** is a finite, usually small, set of symbolic constants.

```
domain Mode {STANDBY, WORKING, FAILED, MAINTENANCE}
```

Attributes

Attributes help to specify the variable (and more generally the modeling element).

```
float grainStock(init = 0, unit = "tons");
```

Mandatory for variables

Optional (but often very useful).
Only a string in the current version.

Redeclaration of attributes:

```
attribute main.LandingStage.grainStock.init = 0;
```

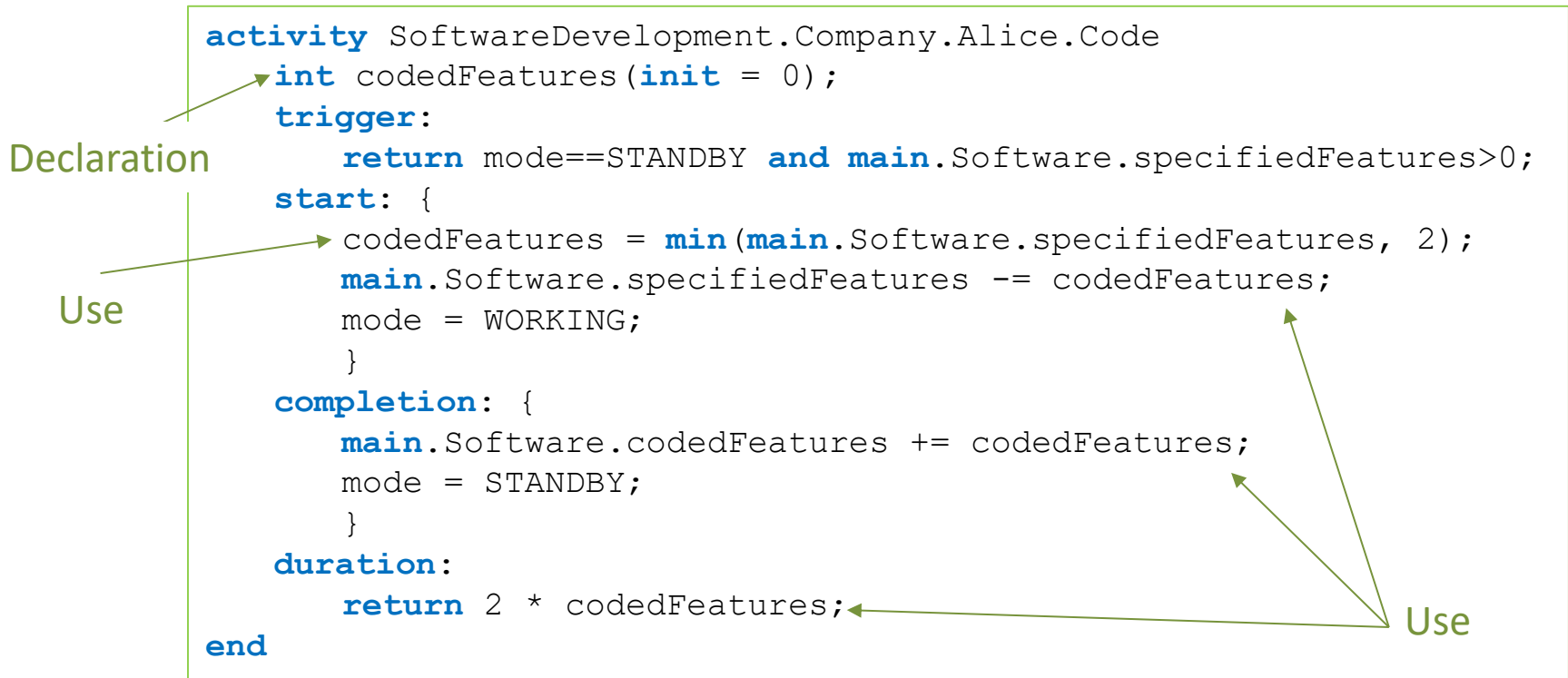
State Variables

State variables are declared with the systems. They are available everywhere in the model.

```
activity SoftwareDevelopment.Company.Alice.Code
  trigger:
    return mode==STANDBY and main.Software.specifiedFeatures>=2;
  start: {
    main.Software.specifiedFeatures -= 2;
    mode = WORKING;
  }
  completion: {
    main.Software.codedFeatures += 2;
    mode = STANDBY;
  }
  duration:
    return 2;
end
```

Activity Variables

Activity variables are declared at the beginning of an activity. Each execution of this activity has its own set of variables.



Routine Variables

Routine variables are local to the block of instructions in which they are declared. Their declaration does not require an `init` attribute.

```
activity SoftwareDevelopment.Company.Alice.Code
  int codedFeatures(init = 0);
  trigger:
    return mode==STANDBY and main.Software.specifiedFeatures>0;
  start: {
    int capacity; ← Declaration
    capacity = rangeDeviate(1, 3);
    codedFeatures = min(main.Software.specifiedFeatures, capacity);
    main.Software.specifiedFeatures -= codedFeatures;
    mode = WORKING;
  }
  completion: {
    main.Software.codedFeatures += codedFeatures;
    mode = STANDBY;
  }
  duration:
    return 2.0;
end
```

Use →

SoftwareDevelopment



LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 4. CONSTANTS, PARAMETERS AND PORTS

Constants and Parameters (1)

Constants and **parameters** can be seen as variables that are set once for all. They are used instead of numerical constants, so to document the models and to make their modification easier.

The EBNF grammar for constant and parameter declaration is as follows.

```
ConstantDeclaration ::=
    'constant' Type Path AttributeList? '=' Expression ';'
ParameterDeclaration ::=
    'parameter' Type Path AttributeList? '=' Expression ';'
```

The expression defining a constant may contain other constants, but not parameters (nor of course variables).

The expression defining a parameter may contain constants and other parameters. A constant or a parameter cannot depend eventually on itself.

The key difference between constants and parameters is that it is possible to modify the value of parameters by passing to assessment tools a file. This is not possible with constants.

Constants and Parameters (2)

```
system SoftwareDevelopment.Company.Alice
  parameter int minimumCapacity = 1;
  parameter int maximumCapacity = 3;
  Mode mode(init = STANDBY);
end

activity SoftwareDevelopment.Company.Alice.Code
  ...
  start: {
    int capacity;
    capacity = rangeDeviate(minimumCapacity, maximumCapacity);
    codedFeatures = min(main.Software.specifiedFeatures, capacity);
    main.Software.specifiedFeatures -= codedFeatures;
    mode = WORKING;
  }
  ...
end
```

SoftwareDevelopment



Ports (1)

As parameters, **ports** are declared together with an expression defining their value. Contrary to parameters however, the latter may involve variables and other ports (under the condition that the port does not depend eventually on itself). The value of the port is updated each time the value of one of variables or ports involved in its definition is changed.

```
PortDeclaration ::=  
    'port' Type Path AttributeList? '=' Expression ';' 
```

Ports are useful to represent quantities that are automatically deduced from other quantities. They make it possible to **propagate information** through the network of components of a system.

Ports (2)

```
system SoftwareDevelopment.Company
...
port int activeWorkers =
    (if Alice.mode==WORKING then 1 else 0)
+ (if Bob.mode==WORKING then 1 else 0);
...
end
```

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 5. OBSERVERS

Observers (1)

The main objective of Σ^{TM} modeling is to assess performance indicators. Performance indicators are numerical quantities. In Σ^{TM} , they are defined via **observers**. Observers are just like real valued ports, except that they cannot be used in expressions.

The EBNF grammar of observer declarations is as follows.

```
ObserverDeclaration ::=
    'observer' Path AttributeList? '=' Expression ';' ;
```

For each observer, the following quantities are continuously updated:

- Its current value;
- Its minimum, maximum and mean values since the beginning of the execution;
- The integral of its value since the beginning of the execution;
- The first date at which it changed of value after the beginning of the execution;
- The number of times it changed of value since the beginning of the execution.

Changes of values are considered only if they last for a strictly positive time.

Observers (2)

```
system SoftwareDevelopment
...
system Software
  parameter int expectedFeatures = 20;
  int specifiedFeatures(init = 0);
  int codedFeatures(init = 0);
  int testedFeatures(init = 0);
end
observer achievementRatio = testedFeatures / expectedFeatures;
end
```

SoftwareDevelopment



LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 6. INSTRUCTIONS

Instructions

The current version of Σ^{TM} provides a limited, yet so far sufficient, set of **instructions**.

```
Instruction ::=
    Skip | Assignment | IfThenElse | While | Return | Block

Skip ::=
    'skip' ';'

Assignment ::=
    Variable ('=' | '+=' | '-=') Expression ';'

IfThenElse ::=
    'if' BooleanExpression 'then' Instruction ('else' Instruction)?

While ::=
    'while' BooleanExpression Instruction

InstructionBlock ::=
    '{' Instruction+ '}'
```

Unit Operation

```
domain State {STANDBY, WORKING, FINISHED, ABORTED}

system Unit
  State state(init = STANDBY);
  parameter float operationTime = 100.0;
  parameter float failureRate = 0.0123;
end

activity Unit.Operate
  float actualTime(init = 0.0);
  trigger:
    return state==STANDBY;
  start: {
    actualTime = min(exponentialDeviate(failureRate), operationTime);
    state = WORKING;
  }
  completion:
    if operationTime<actualTime           // if-then-else instruction
    then state = FINISHED;
    else state = ABORTED;
  duration:
    return actualTime;
end
```

Service Line

```
system ServiceLine
  parameter float minimumServiceTime = 3;
  parameter float maximumServiceTime = 10;
  parameter float expectedServiceTime = 5;
  int waitingClients(init = 10);
  bool serving(init = false);
end

activity ServiceLine.ServeClients
  float serviceTime(init = 0);
  trigger:
    return not serving;
  start: {
    while waitingClients>0 {
      serviceTime += triangularDeviate(minimumServiceTime,
                                       maximumServiceTime, expectedServiceTime);
      waitingClients -= 1;
    }
    serving = true;
  }
  completion:
    skip;
  duration:
    return serviceTime;
end
```

ServiceLine



LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 7. EXPRESSIONS

Expressions (1)

Σ^{TM} provides a large set of **expressions**.

Literal constants:

- Boolean constants: true, false
- Integers: 42, -33
- Floating point numbers: 1.23, 2.7e-8
- Symbolic constants: STANDBY, WORKING (must be declared in a domain)
- Strings: "Data/DailyProduction.csv"

Reference to variables, constants and parameters

Boolean expressions:

- and, or, not

Inequalities:

- ==, !=, <, <=, >, >=

Expressions (2)

Arithmetic expressions:

- `+, -, *, /` (the division of two integers returns always a float)

Builtin expressions:

- `min, max, abs, div, mod, exp, log...`

Conditional expression:

- `if ... then ... else ...`

Random deviates (see section 8)

Connections to external data sets (see section 9)

Time expressions

- `missionTime(), currentTime()`

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 8. RANDOM DEVIATES

Uncertainties

Many phenomena at stake in engineering are not known with certainty, e.g. duration of a task, time to failure of a pump...

There are however two types of uncertainties*:

- **Epistemic uncertainty**: we do not know why something is happening.
- **Aleatory uncertainty**: we do not when something will happen, but we can make statistics or at least informed guesses the phenomenon.

If there is not much we can do about epistemic uncertainty (garbage-in, garbage-out), we can introduce a **controlled randomness** in models to deal with aleatory uncertainties.

In Σ^{TM} , this is implemented by means of special expressions called **random deviates**.

(*) There is arguably a third type of uncertainty resulting from our limited computational capacities, but this goes beyond the scope of this lecture.

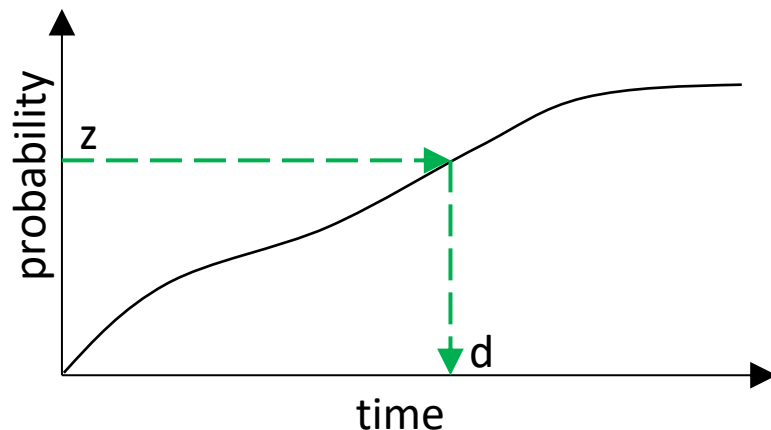
Definition

A **probability distribution** is a non decreasing function from $\mathbb{R}^+$ into $[0, 1]$.

Intuitively, a probability distribution associates to each time t , the probability that a certain event is realized before t .

A **random deviate** is the inverse of a probability distribution.

Intuitively, a random deviate picks up a number z uniformly at random in $[0, 1]$ and returns the least time d such that the probability that a certain is realized before d is greater or equal to z .

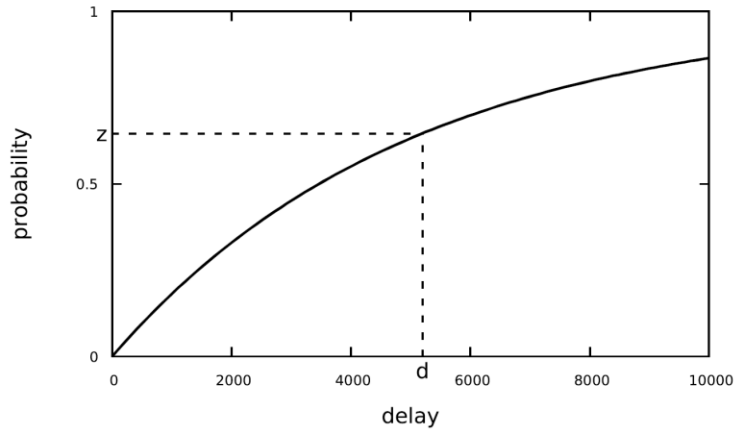


Note that the randomness is doubly controlled:

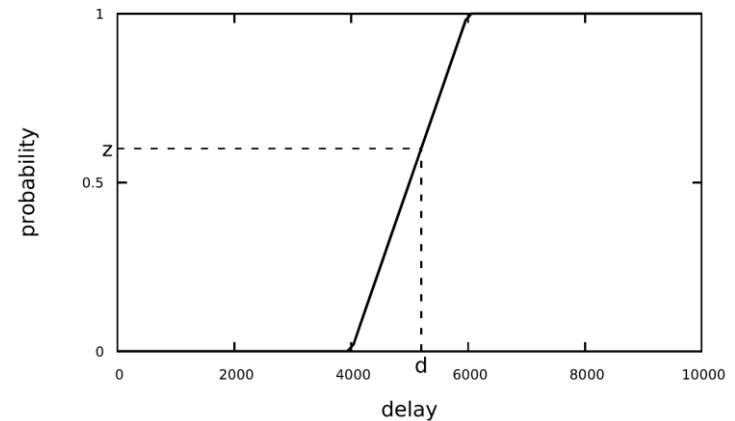
- First, by using **known probability distributions**.
- Second, by using **algorithmic random generator** to pick up values of z .

Probability Distributions

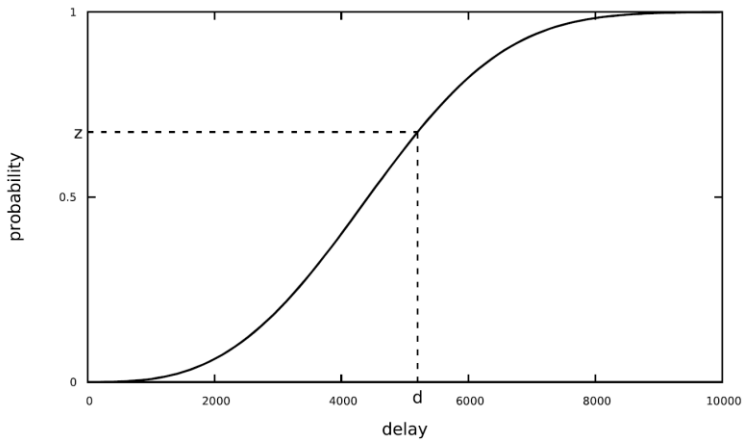
Exponential



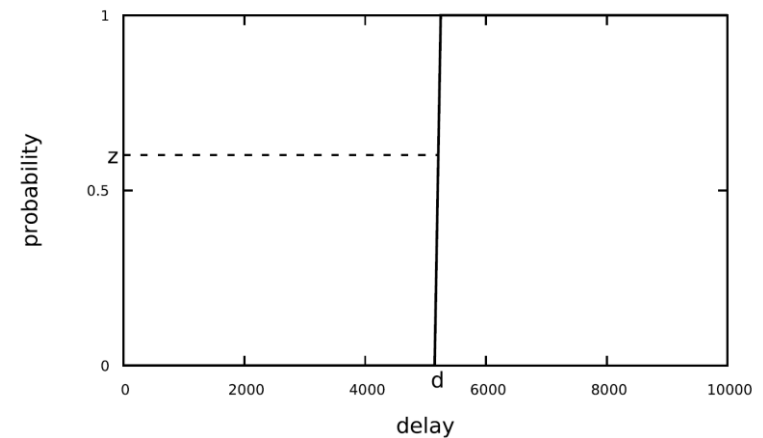
Uniform



Weibull



Dirac



Random Deviates

Random deviates available in the current version of Σ^{TM} :

```
ParametricRandomdeviate ::=  
  'uniformDeviate' '(' Expression ',' Expression ')'  
| 'triangularDeviate' '(' Expression ',' Expression ',' Expression ')'  
| 'normalDeviate' '(' Expression ',' Expression ')'  
| 'lognormalDeviate' '(' Expression ',' Expression ')'  
| 'exponentialDeviate' '(' Expression ')'  
| 'WeibullDeviate' '(' Expression ',' Expression ')'  
| 'rangeDeviate' '(' Expression ',' Expression ')'
```

```
uniformDeviate(3.1, 3.6)  
exponentialDeviate(1.23e-6)  
rangeDeviate(0, 10)
```

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 9. EXTERNAL DATA SETS

Data-Driven Modeling

Σ^{TM} models often involve **external data sets**. Referring to external data sets has at least two major interests:

- It makes it possible to embed complex field data in the models, which eases the **calibration** of the latter (to check their validity against real life).
- It makes the models **parametric** in this sense that we can test them with different data sets.

The current version of Σ^{TM} provides primitives to read data into:

- CSV files
- Excel[®] Spreadsheets.

CSV Files

CSV stands for “Comma Separated Values”. **CSV Files** are text files in which encode **tabular data**. Each line represents a row, columns are separated with commas. Σ^{TM} makes use of the tabulation as a separator, therefore of TSV files, although it uses the “.csv” extension.

	MAR	JUN	SEP	DEC		
GOOG	104.000000		120.970001		131.850006	140.929993
AMZN	103.290001		130.360001		127.120003	151.940002
META	211.715363		286.675842		299.891815	353.584839
APPL	164.024460		193.207016		170.766830	192.284637
MSFT	285.953064		338.506226		314.528809	375.345886

GAFAMStockPrices



Data Set Primitives

```
EmpiricalDistribution ::=
  'empiricalDistribution' '(' String ',' Interpolation ',' Expression ')'

EmpiricalDeviate ::=
  'empiricalDeviate' '(' String ',' Interpolation ')'

Interpolation ::= 'step' | 'linear'

CSVTableCall ::=
  'table' '(' String ',' Expression, Expression ')'

SpreadsheetCall ::=
  'spreadsheet' '(' String ',' Expression, Expression, Expression ')'

String = "[^"]+"

```

```
empiricalDistribution("../Data/CrudeOilPrice.csv", linear, currentTime())
empiricalDeviate("../Data/TimeToFailure.csv", linear)
table("../Data/StockPrices.csv", 2, 4)
spreadsheet("../Data/StockPrices.xlsx", "StockPrices 2023", "D", "2")

```

LECTURE 3. DIVING DEEPER INTO Σ^{TM}

SECTION 10. COMMENTS

Comments

Aside the ellipsis "...", Σ^{TM} provides two types of **comments**:

- Single line comments: //
- Multiline comments: /* ... */

```
/*  
 * Description of Alice who is in charge of both specifying and coding  
 * the software.  
 */  
system SoftwareDevelopment.Company.Alice  
    parameter int minimumCapacity = 1; // features per two weeks sprint  
    parameter int maximumCapacity = 3; // idem  
    Mode mode(init = STANDBY);  
end
```

Comments are useful. But remember, that the best comment is to give significant names to modeling elements.