SFWRENG / MECHTRON 3K04 Software Development

This Week: Requirements & Validation

McMaster University | Engineering
Computing & Software

September 15th, 2025

Dr. Angela Zavaleta Bernuy

Do you remember what we talked about last week?

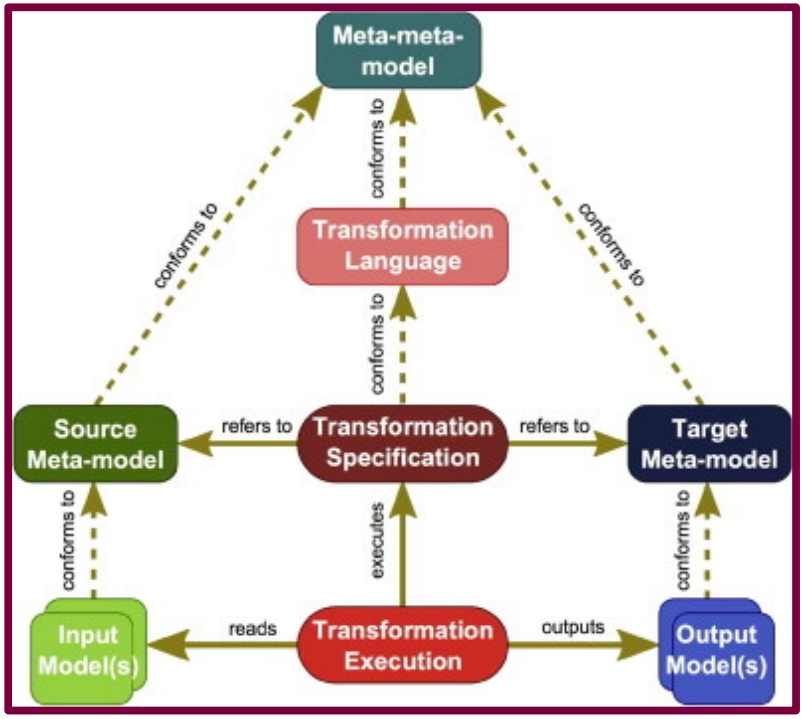
Let’s see...

Modern Trends in SDLC

- Agile:
  - Meant for smaller teams and smaller development cycles
  - Some people are distrustful of how Agile can be used to handle safety critical engineering
- Waterfall
  - Meant for bigger teams and bigger projects
  - Some people dislike how long it takes to develop using waterfall
  - Tried and tested for development of critical components
- We need a methodology that can fit in the SDLC to make development of safety critical systems efficient
  - Model-Driven Engineering!

Model-Driven Engineering (MDE)

- It makes models the main artifact of development instead of raw code
  - Code can be generated automatically from these models
  - E.g., we can transform a UML diagram into Java classes, or a state machine model into embedded system code ;)
- Goal: raise the level of abstraction so developers think in **domain concepts** rather than low-level programming
- Reduce manual coding and relies on code generation and automation tools
- **Is MDE an SDLC?**



https://www.sciencedirect.com/topics/computer-science/model-driven-engineering

MDE in Safety-critical Systems

- **Assurance**: Making statements about properties of a system and demonstrate that we have some adequate level of confidence that statements are true.
- In safety-critical systems, we want to *assure* that the system is safe, secure and dependable
- Traditionally, assurance was implicit
  - Developers would provide assurance by providing the documentation and test results of the system
- With MDE, assurance can be explicit
  - We will talk about this throughout the course

Takeaways!

- A number of SDLC models are being used and have been transformed across the years
  - Our focus is safety critical systems, for which spiral and V model are used often
  - In particular, the spiral model emphasizes the iterative nature of hazard analysis
  - Of course, we can use other SDLC models, but keep hazard/risk analysis in mind
- MDE is prevalent in many industries, specially for safety-critical systems
- Explicit assurance is needed to ensure the systems we develop are safe, secure and dependable

Now, Requirements!

Requirements & Validation

- If we don't get the requirements right, it does not matter how good we build our system.
- The world of *requirements*, can be divided into a few parts:
  - **Eliciting** requirements: how do we **capture** the requirements
  - **Documenting** requirements: how do we **keep track** of the requirements
    - **Formal specifications**: well-defined syntax and semantics (usually mathematical)
    - **The 4 variable model**: relationship between requirements and design
  - How do we know if we elicit and document requirements **correctly**?
- *Validation*: determining whether or not we got the *right* requirements
  - Not to be confused with *verification*, which is determining whether or not we built the system *right* (i.e., if it complies with the requirements)
  - We often refer to Validation and Verification as V & V.

Documenting Requirements: Natural Language

- Most requirements are written in natural language
- Often they consist of a succession of points or paragraphs that do not collect all related information
- Each paragraph usually contains multiple items
- What are some **problems** with using natural language?
  - It can be ambiguous, imprecise, inconsistent…
- What do we **require** from requirements?
  - For a given input, what should the output be? I.e. how should the system behave?
  - Recall the continuity property!

Clear, unambiguous, understandable?

Example: specification fragment for a word-processor

> Selecting is the process of designating areas of the document that you want to work on. Most editing and formatting actions require two steps: first you select what you want to work on, such as text or graphics; then you initiate the appropriate action.

Can an area be scattered?

Consistent?

Example: specification fragment for a word-processor

> The whole text should be kept in lines of equal length. The length is specified by the user. Unless the user gives an explicit hyphenation command, a carriage return should occur only at the end of a word.

What if the length of a word exceeds the length of the line?

Precise, unambiguous, clear?

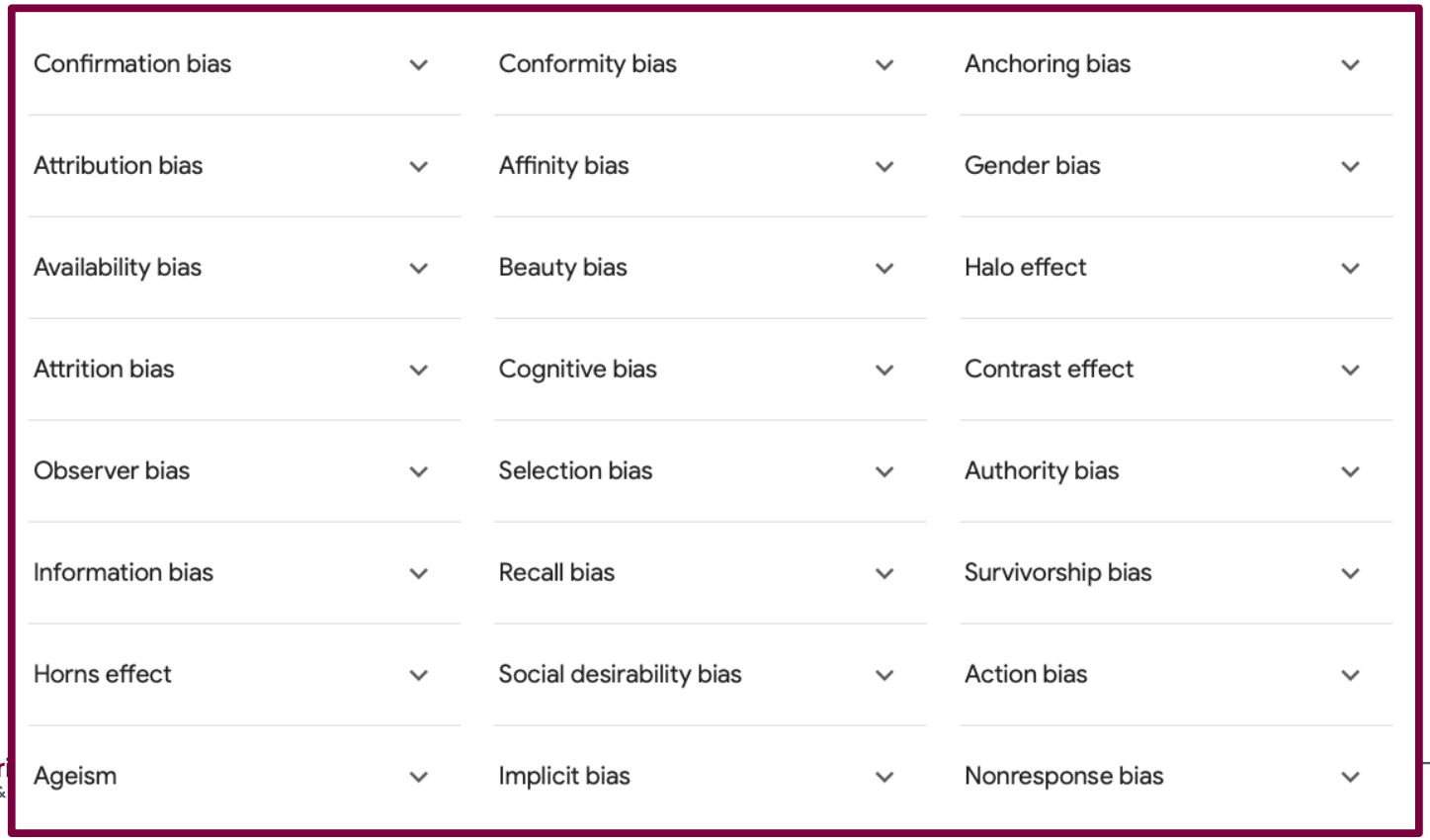
Example: specification fragment for a spacecraft (safety-critical systems)

> The message must be triplicated. The three copies must be forwarded through three different physical channels. The receiver accepts the message on the basis of a two-out-of-three voting policy.

Can a message be accepted as soon as we receive 2 out of 3 identical copies of message or do we need to wait for receipt of the 3rd?

Bias in Requirements

When we google the word bias, these is what we get:



Bias in Requirements

- Biases exist everywhere
  - And these lead to assumptions!
- They are both conscious and unconscious, and help us to make sense of the requirements
  - Sometimes, poor sense of the requirements
- It is critically important when writing or evaluating requirements to be aware of biases
  - Be aware of how your bias as an engineer may inform decisions around requirements
- Diversity in opinions and experiences are extremely helpful to avoid *group think* and mitigate biases in the requirements

Formal Specifications

- Specifications are central to many aspects of software engineering
- A *specification* is just a precise description of behaviour
- *Requirements specifications* describe required behaviour in a system
- *Design specifications* describe the behaviour prescribed in a design

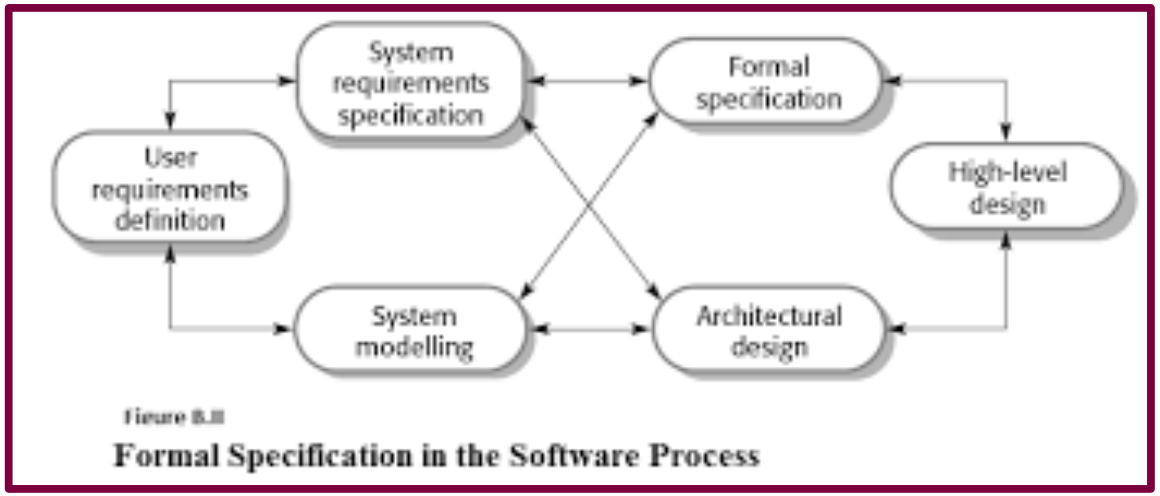
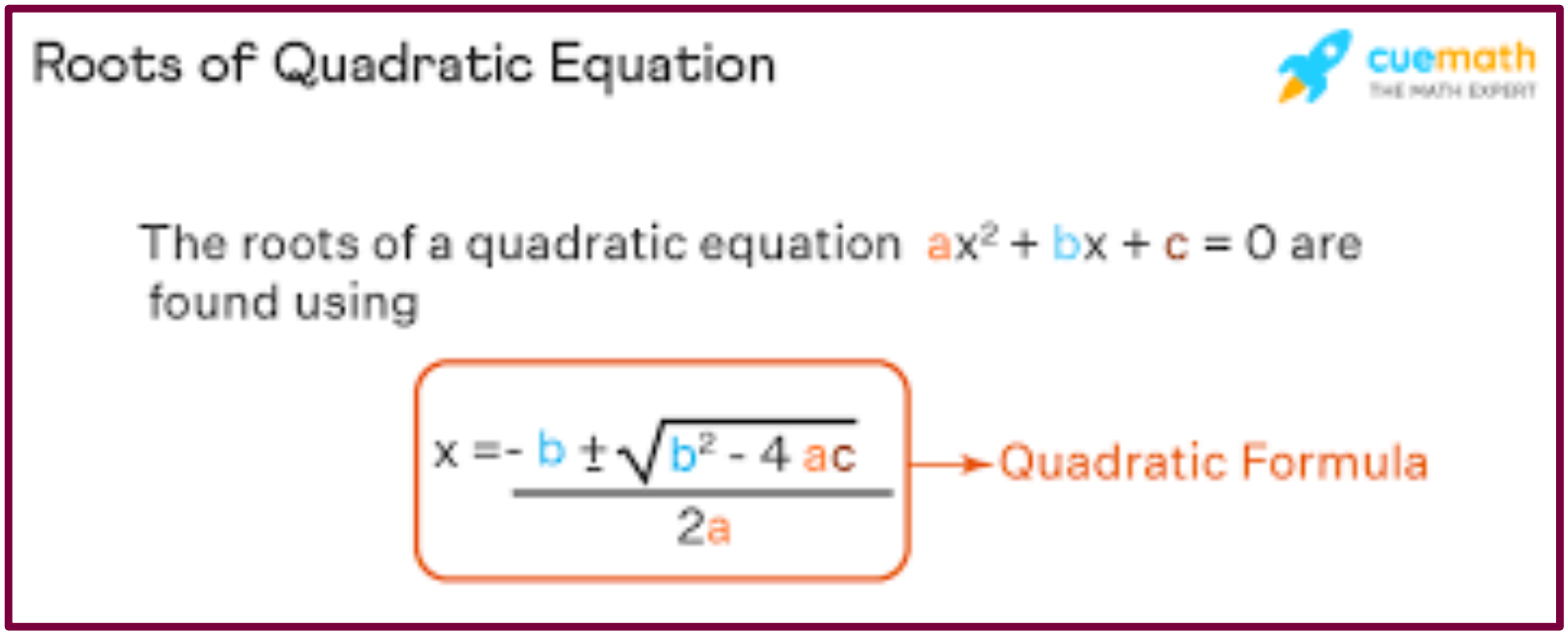


Figure 8.8

Formal Specification in the Software Process

https://www.dline.info/pca/fulltext/v5n2/pcav5n2_2.pdf

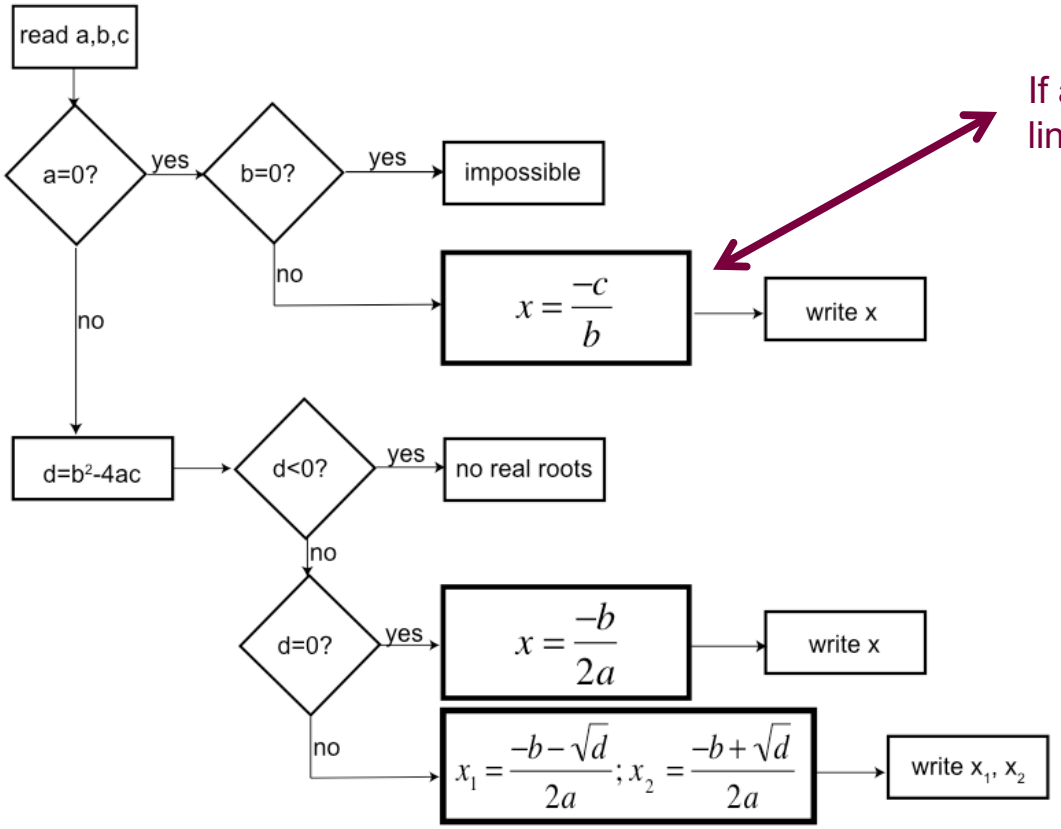
Example: Solving a Quadratic Equation



https://www.cuemath.com/algebra/roots-of-quadratic-equation/

Example: Flowchart

$ax^2+bx+c=0$



If a=0, it's a linear equation!

Floating-point problems?

Example: Pseudocode

```
if (a=0 & b=0) then solution_caption := “Ill-conditioned”
else if (b≠0) then
    solution_caption := “One root”; x1 := -c/b
else
    d := b2-4ac
    if (d<0) then solution_caption := “No real roots”
    else if (d=0) then
            solution_caption := “One real root”; x1 = -b/(2a)
    else
            solution_caption := “Two roots”
            if (b≥0) then x1 := (-b-√d)/(2a)
            else x1 := (-b+√d)/(2a)
            x2 := c/(ax1)
```

Is this a *good* specification?
Is it easy to understand?
Is this a **requirement** or is it a **design** specification?
What is the difference?

Example: Solving a Quadratic Equation

- Is this a good specification?
  - Not really. It is too detailed. It is mixing up *what* needs to be achieved with *how* it should be done.
- Is it easy to understand?
  - Yes, for engineers, but not for non-technical people
- Is this a requirement or is it a design specification?
  - Design! It explains how to compute the solution
- What is the difference?
  - *Requirements specifications* describe required behaviour in a system (WHAT)
    - *"The calculator shall determine and report the number of real solutions of a quadratic equation and provide their values if they exist."*
  - *Design specifications* describe the behaviour prescribed in a design (HOW)
    - *"If discriminant d < 0 then return 'no real roots'. Else compute roots using x = (-b ± √d) / (2a)."*

Example: Tabular Expression

			Solution caption	x_1	x_2
$a = 0$	$b = 0$		Ill defined	—	—
	$b \neq 0$		One root	$-\frac{c}{b}$	—
$a \neq 0$	$b^2 - 4ac < 0$		No real roots	—	—
	$b^2 - 4ac = 0$		One root	$-\frac{b}{2a}$	—
	$b^2 - 4ac > 0$	$b \geq 0$	Two roots	$\frac{-b-\sqrt{b^2-4ac}}{2a}$	$\frac{c}{ax_1}$
		$b < 0$	Two roots	$\frac{-b+\sqrt{b^2-4ac}}{2a}$	$\frac{c}{ax_1}$

What are the differences?

	Flowchart	Pseudocode	Tabular Expression
Differences	Shows the control flow graphically, visualize decision paths	Shows the logic in a programming-like style	Shows the conditions and outcomes clearly and compactly
Strengths	Easier for many people to follow, highlights paths and edge cases	Close to implementation, unambiguous for developers	Excellent for coverage, you need to list all possible conditions and outcomes. Compact, easy to check consistency
Weaknesses	It can get messy for complex logic	Looks like design, not requirements. Hard for non-technical stakeholders	Can grow big if many conditions

Example: Simple Control Interaction – Pseudocode

The operator in a real-time application requests or cancels a setpoint, **c_sp**, by pressing momentary pushbuttons. There are two pushbuttons, called **m_pbON** and **m_pbOFF**. They can take on the values **e_pressed** and **e_notPressed**. We specify the required behaviour as follows:

```
if (m_pbON = e_pressed ^ m_pbOFF = e_notPressed)
        c_sp = e_requested
else if (m_pbON = e_notPressed ^ m_pbOFF = e_pressed)
        c_sp = e_cancelled
```

Any problems with this?
What is both push buttons are not pressed? Or are pressed at the same time?

No specified behaviour! Then, there would not be a change in the current **c_sp** value.
It won't be continuous!

Example: Tabular Expression

Clear, unambiguous and complete!

Condition	*Result* **c_sp**
(m_pbON = e_notPressed) & (m_pbOFF = e_notPressed)	No Change
(m_pbON = e_notPressed) & (m_pbOFF = e_pressed)	e_cancelled
(m_pbON = e_pressed) & (m_pbOFF = e_notPressed)	e_requested
(m_pbON = e_pressed) & (m_pbOFF = e_pressed)	e_requested

Tabular Expressions

- A general, very simple, tabular expression:

Condition	*Result* **function name**
condition 1	result 1
condition 2	result 2
…	…
condition n	result n

- This is equivalent to:
  - if (condition 1) then function name = result 1
  - else if (condition 2) then function name = result 2
  - else if …
  - else if (condition n) then function name = result n
- Disjointness: No conditions in the table overlaps (cond i $\wedge$ cond j $\Leftrightarrow$ FALSE $\forall$i,j = 1,..,n, i $\neq$ j)
- Completeness: The conditions together cover all possible input cases (cond 1 $\vee$ cond 2 $\vee$ … $\vee$ cond n $\Leftrightarrow$ TRUE)

Semi-Formal Specifications

- On one hand, we can have informal specifications (natural language), while on the other, we can have formal specifications (predicate logic, tabular expressions, data flow diagrams, etc.)
- Semi-formal specifications fall somewhere in between
- For example, the Gherkin specification languages (https://cucumber.io/docs/gherkin/reference)

Semi-Formal Specifications - Gherkin

- Gherkin combines natural language specification with some keywords to give it a more rigorous structure compared to free-hand writing
- Three main keywords are used: GIVEN, WHEN, THEN
  - **GIVEN**: used to describe the initial context of the system
    - The *scene* of the scenario. It is typically something that happened in the *past*. (Preconditions)
  - **WHEN**: used to describe an event, or an *action*.
    - This can be a person interacting with the system, or it can be an event triggered by another system. (Events)
  - **THEN**: used to describe an *expected* outcome, or *result*. (Postconditions)

Example: Semi-Formal Specifications - Gherkin

- Gherkin specification typically best used to specify scenarios

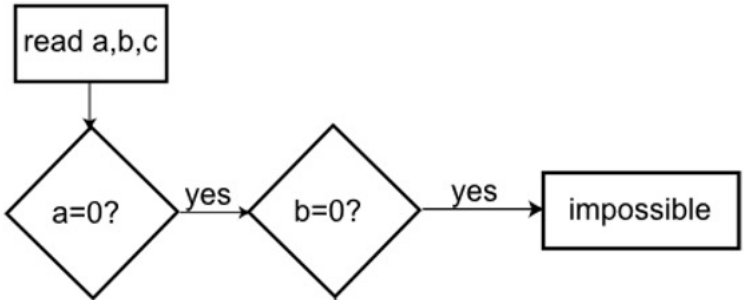
GIVEN three variables a, b, c

AND

GIVEN a = 0 and b=0

WHEN calculating the roots of a quadratic equation

THEN the resulting roots are impossible to determine



- It can also be written using **AND**: used to cojoin multiple **GIVEN/WHEN/THEN** statements

Example: Semi-Formal Specifications - Gherkin

- Can also be written using **BUT**: used to conjoin multiple **GIVEN/WHEN/THEN** statements, usually a constraint or an opposing specification

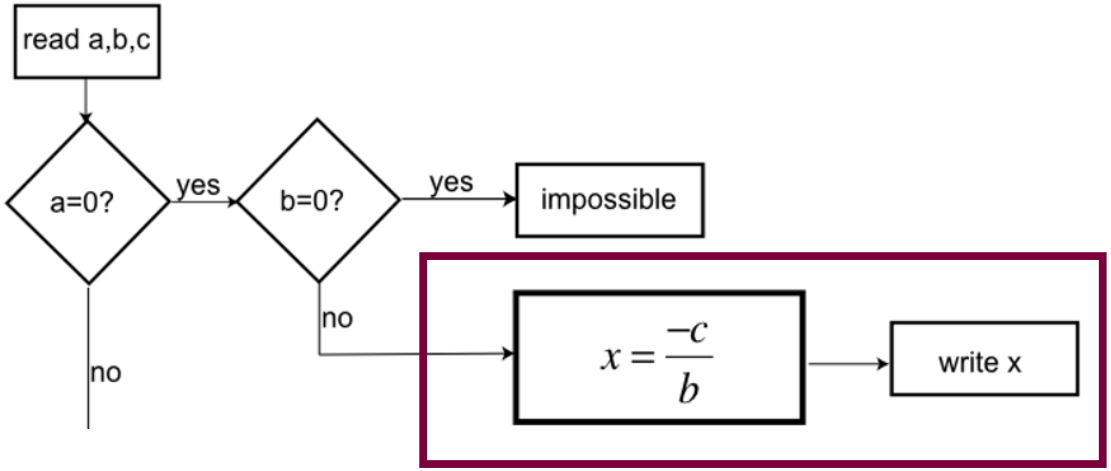
GIVEN three variables a, b, c

AND a = 0

BUT b ≠ 0

WHEN calculating the roots of a quadratic equation

THEN the resulting root x = -c/b



Exercise: Semi-Formal Specifications - Gherkin