DTU

2024-05-16

# Brugervenlige formelle metoder

IDA IT

# Brugervenlige formelle metoder

Frederik Krogsdal Jacobsen & Jørgen Villadsen

# Om os

| Frederik | Jørgen |
|---|---|
| Bevisassistenter | Bevisassistenter |
| Logik | Logik |
| Typeteori | Typeteori |
| Distribuerede systemer | Kunstig intelligens |





| Frederik | Jørgen |
|---|---|
| Bevisassistenter | Bevisassistenter |
| Logik | Logik |
| Typeteori | Typeteori |
| Distribuerede systemer | Kunstig intelligens |

1. Med kunstig intelligens menes primært multi-agent systemer, og ikke machine learning

DTU

# Agenda

1. Hvad er en bevisassistent?
2. Traditionelle anvendelser
3. Pause
4. Formelle metoder i almindelig software
5. Spådomme





1. Hvad er en bevisassistent?
2. Traditionelle anvendelser
3. Pause
4. Formelle metoder i almindelig software
5. Spådomme

# Matematik = programmering

int
string





int
string

1. Venstre: grundtyper

## Matematik = programmering

List<T>
$'$a tree

int
string

(char $\to$ char) $\to$ string $\to$ string

$\{$int x, $\;$ x $< 5\}$
$\{$string x, $\;|$x$| = 10\}$





List<T>
'a tree
int
string
(char → char) → string → string
{int x, x < 5}
{string x, |x| = 10}

1. Venstre: grundtyper
2. Øverst: System Weak Omega (generiske typer)
3. Midterst: System F (polymorfi)
4. Nederst: dependent types

DTU

# Matematik = programmering

List<T>
'a tree

int
string

(char → char) → string → string

int m → {int p, m < p ∧ prime(p)}

$\{\text{int } x,\ x < 5\}$
$\{\text{string } x,\ |x| = 10\}$





List<T>
'a tree
int
string
(char → char) → string → string
int m → {int p, m < p ∧ prime(p)}
{int x, x < 5}
{string x, |x| = 10}

1. Venstre: grundtyper
2. Øverst: System Weak Omega (generiske typer)
3. Midterst: System F (polymorfi)
4. Nederst: dependent types
5. Højre: Calculus of Constructions
6. Alle krav/specifikationer kan udtrykkes som en type

# Matematik = programmering

| Matematik | = | Programmering |
|---|---|---|
| Implikation | = | Funktion |
| Konjunktion | = | Tuple |
| Disjunktion | = | Tagged union |
| Sætninger | = | Typer |
| Beviser | = | Programmer |






| Matematik | = | Programmering |
|---|---|---|
| Implikation | = | Funktion |
| Konjunktion | = | Tuple |
| Disjunktion | = | Tagged union |
| Sætninger | = | Typer |
| Beviser | = | Programmer |


1. Mere info: `https://en.wikipedia.org/wiki/Curry-Howard_correspondence`

# Matematik + programmering

| Matematik | Programmering |
| --- | --- |
| Implikation | Funktion |
| Konjunktion | Tuple |
| Disjunktion | Tagged union |
| Sætninger | Typer |
| Beviser | Programmer |

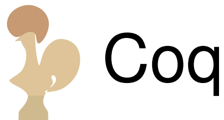


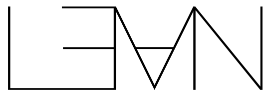


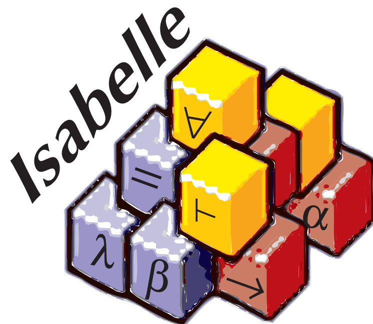
Matematik Programmering
Implikation Funktion
Konjunktion Tuple
Disjunktion Tagged union
Sætninger Typer
Beviser Programmer

1. I stedet for at insistere på at programmering og logik er det samme kan man også holde dem adskilt: simple type theory
2. Simple type theory er lettere at forstå og opfører sig mere som almindelig matematik end Calculus of Constructions
3. Man løber ikke så let ind i teoretiske begrænsninger som i calculus of constructions

# Bevisassistenter


Coq










1. Coq, Lean, Agda: programmer som beviser/typer som logik
2. Isabelle: programmer og "almindelig" matematik
3. Eksempel med McCarthy’s 91-funktion i Isabelle: returnerer 91 for alle input indtil 100, og derefter n - 10

# Eksempel: McCarthy's 91-funktion i Isabelle

As the field of Formal Methods advanced, this example appeared repeatedly in the research literature. In particular, it is viewed as a "challenge problem" for automated program verification.
`https://en.wikipedia.org/wiki/McCarthy_91_function`

$$M(n) = \begin{cases} n - 10, & \text{if } n > 100 \\ M(M(n+11)), & \text{if } n \leq 100 \end{cases}$$

```
theory McCarthy imports Main begin

function M :: ‹int ⇒ _› where ‹M n = (if n > 100 then n - 10 else M (M (n + 11)))› \<proof>

termination \<proof>

theorem ‹M n = (if n > 100 then n - 10 else 91)› \<proof>

end
```





1. Fra start 1970'erne

# Eksempel: McCarthy's 91-funktion i Isabelle

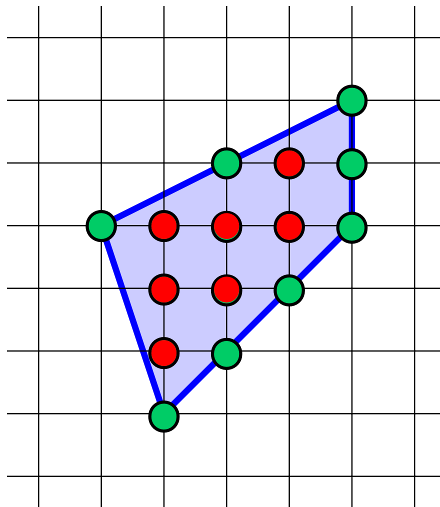
```
theory McCarthy imports Main begin

function M :: ‹int ⇒ _› where ‹M n = (if n > 100 then n - 10 else M (M (n + 11)))›
  by simp_all

termination
proof
  let ?R = ‹measure (λn. nat (101 - n))›
  show ‹wf ?R› ..
  fix n :: int
  assume ‹¬ n > 100›
  moreover from this show ‹(n + 11, n) ∈ ?R›
    by simp
  assume ‹M_dom (n + 11)›
  moreover have ‹n - 10 ≤ M n› if ‹M_dom n› for n
    using that by induct (force simp: M.psimps)
  ultimately show ‹(M (n + 11), n) ∈ ?R›
    by force
qed

theorem ‹M n = (if n > 100 then n - 10 else 91)›
  by (induct n rule: M.induct) simp

end
```

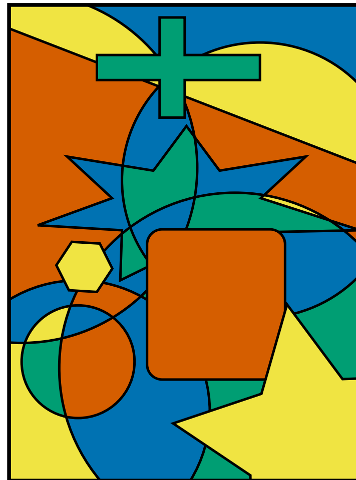




1. Backup slide

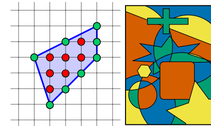
# Intermezzo: matematiske beviser





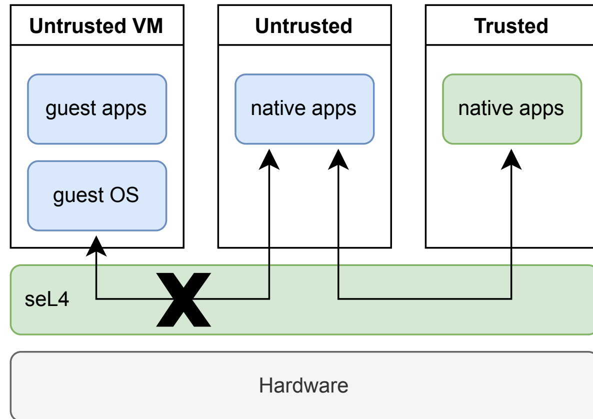






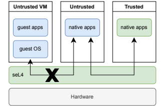
1. Både almindelig matematik og beviser, som kun kan lade sig gøre med en computer
2. Pick's sætning: areal = indvendige punkter + (kantpunkter / 2) - 1
3. Four colour theorem: alle "kort" kan farvelægges med fire farver således at ingen kanter med samme farve støder op til hinanden. Beviset er så indviklet, at det kun kan lade sig gøre med en computer

# Case study: seL4



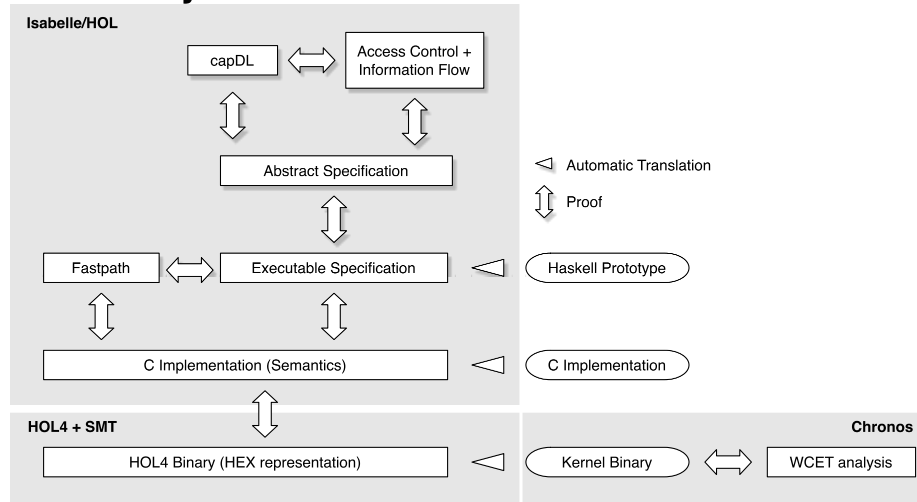




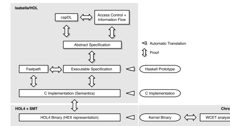
1. Verificeret mikrokerne
2. Til at adskille forskellige processer på den samme maskine
3. Bevis: Kommunikation kan kun ske igennem specifikke kanaler
4. Bevis: Der er ingen sikkerhedshuller i operativsystemets kerne
5. For eksempel er ingen eskalering af privilegier mulig
6. Hvordan kommer man i gang? Flyt langsomt dele af applikationen fra usikker virtuel maskine (Linux) til "native" applikationer (seL4)

# Case study: seL4







1. Haskell-prototype af mikrokernen (til simulering) oversættes til specifikation i Isabelle
2. capDL: Hvilke applikationer har adgang til hvad (hukommelseslayout, andre applikationer)
3. Specifikationen for det overordnede system genereres ud fra en beskrivelse af arkitekturen i systemet og de krav til access control man opstiller
4. C-koden (manuelt skrevet) bevises korrekt i forhold til specifikationen
5. Den kompilerede maskinkode verificeres også ift. korrekthed og understøtter tidsanalyse
6. Andre lag kan bygges ovenpå (eller under) og integreres med beviset for seL4
7. Mere information: `https://sel4.systems/`
8. Andre projekter:
   - CompCert: Verificeret C compiler i Coq (`https://compcert.org/`)
   - Sail: Verificering af processordesign med Isabelle (`https://github.com/rems-project/sail`)

# Pause





└─Pause

# Fra test til beviser

Enkelte eksempler $\longrightarrow$ Klasser af eksempler $\longrightarrow$ Beviser

| | | |
|---|---|---|
| sort $[1, 4, 3] = [1, 3, 4]$ | $\lvert$sort xs$\rvert = \lvert$xs$\rvert$ | $\forall$x $\in$ xs. x $\in$ sort xs |
| add 2 2 $=$ 4 | for x in [1..1000] {add x x = 2 * x} | $\forall$x y. add x y $=$ x $+$ y |

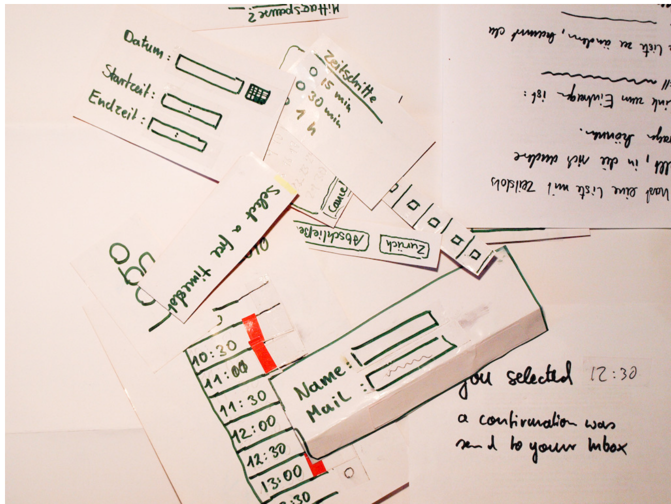




Enkelte eksempler ⟶ Klasser af eksempler ⟶ Beviser
sort [1, 4, 3] = [1, 3, 4] |sort xs| = |xs| ∀x ∈ xs. x ∈ sort xs
add 2 2 = 4 for x in [1..1000] {add x x = 2 * x} ∀x y. add x y = x + y

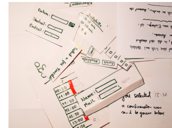
1. Almindelige test: enkelte eksempler på input/output
2. Property-based testing: regler, som kan bruges til at generere mange eksempler
   - Enkelte eksempler vil altid udspringe af regler, så hvorfor ikke skrive reglerne ned?
   - Samtidig videregiver man hvad den overordnede tanke er og ikke kun den enkelte test
3. Beviser:
   - Reglerne vil altid overholdes
   - Property-based testing virker ikke så godt hvis der er et stort eller kompliceret tilstandsrum
4. Man behøver ikke altid gå hele vejen! Men tankegangen er altid brugbar
5. Selv hvis man ikke beviser noget kan man stadig skrive sine antagelser ned
6. Man kan starte med at bevise at antagelserne ikke er i modstrid med hinanden

# Validering af idéer og antagelser





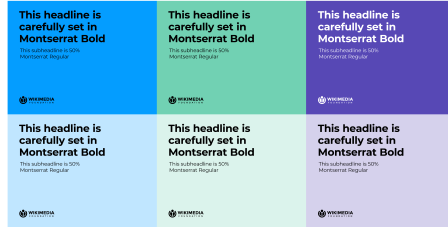


1. Prototyping: skriv et program og en egenskab
2. Brug quickcheck (også tilgængeligt i almindelige sprog)
3. Brug nitpick (mere generelt end hvad man kan få i de fleste sprog)
4. Påbegynd et bevis og find "åbenlyse" fejl
5. Fordel: man kan benytte matematiske koncepter direkte (f.eks. mængder osv.)
6. Fordel: man kan mocke en komponent simpelthen ved bare at beskrive dens egenskaber som assumptions uden nogen implementering (induktiv definition eller locale)

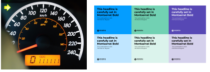
DTU

# Hvad kan man ikke bruge formelle metoder til?







1. Ikke-funktionelle krav er meget svære:
   - Hastighed/performance
   - Grafik og UX er i de fleste tilfælde umuligt
2. Man kan aldrig undgå "fejl 40"
3. Low-level systemer kræver mange detaljer, som kan være svære at simplificere

# Hvor er formelle metoder et godt match?

## Juni 2014

In this run, **a single network partition isolating a primary node caused the loss of over 90% of acknowledged writes.** Of 619 documents inserted, 538 returned successful, but only 54– 10%–of those documents appeared in the final read. The other 484 were silently discarded.

Remember, this is the one kind of network failure Elasticsearch was designed to withstand.





Juni 2014

In this run, **a single network partition isolating a primary node caused the loss of over 90% of acknowledged writes.** Of 619 documents inserted, 538 returned successful, but only 54– 10%–of those documents appeared in the final read. The other 484 were silently discarded.

Remember, this is the one kind of network failure Elasticsearch was designed to withstand.

1. Elasticsearch bruger deres egen algoritme til at håndtere distribueret konsensus
2. Distribuerede systemer er svære at ræsonnere uformelt om
3. Model checking kan ikke skaleres til udtømmende beviser, og mange fejl sker kun i meget specifikke situationer
4. Safety: Hvad må aldrig ske
5. Liveliness: Hvad skal ske på et tidspunkt
6. Længere eksempel med Isabelle: `https://martin.kleppmann.com/2022/10/12/verifying-distributed-systems-isabelle.html`
7. Mere info om fejl i Elasticsearch: `https://aphyr.com/posts/323-jepsen-elasticsearch-1-5-0`

# Hvor er formelle metoder et godt match?

## September 2014

**AeroNotix** commented on Sep 1, 2014

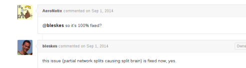
**@bleskes** so it's 100% fixed?

**bleskes** commented on Sep 1, 2014 Owner

this issue (partial network splits causing split brain) is fixed now, yes.








1. Elasticsearch bruger deres egen algoritme til at håndtere distribueret konsensus
2. Distribuerede systemer er svære at ræsonnere uformelt om
3. Model checking kan ikke skaleres til udtømmende beviser, og mange fejl sker kun i meget specifikke situationer
4. Safety: Hvad må aldrig ske
5. Liveliness: Hvad skal ske på et tidspunkt
6. Længere eksempel med Isabelle: `https://martin.kleppmann.com/2022/10/12/verifying-distributed-systems-isabelle.html`
7. Mere info om fejl i Elasticsearch: `https://aphyr.com/posts/323-jepsen-elasticsearch-1-5-0`

# Hvor er formelle metoder et godt match?

## April 2015

Elasticsearch 1.5.0 still loses data in every scenario tested. You can lose documents if:

- The network partitions into two intersecting components
- Or into two discrete components
- Or if even a single primary is isolated
- If a primary pauses (e.g. due to disk IO or garbage collection)
- If multiple nodes crash around the same time

1





April 2015
Elasticsearch 1.5.0 still loses data in every scenario tested. You can lose documents if:
- The network partitions into two intersecting components
- Or into two discrete components
- Or if even a single primary is isolated
- If a primary pauses (e.g. due to disk IO or garbage collection)
- If multiple nodes crash around the same time

1. Elasticsearch bruger deres egen algoritme til at håndtere distribueret konsensus
2. Distribuerede systemer er svære at ræsonnere uformelt om
3. Model checking kan ikke skaleres til udtømmende beviser, og mange fejl sker kun i meget specifikke situationer
4. Safety: Hvad må aldrig ske
5. Liveliness: Hvad skal ske på et tidspunkt
6. Længere eksempel med Isabelle: `https://martin.kleppmann.com/2022/10/12/verifying-distributed-systems-isabelle.html`
7. Mere info om fejl i Elasticsearch: `https://aphyr.com/posts/323-jepsen-elasticsearch-1-5-0`

# Hvor er formelle metoder et godt match?

## Marts 2019

Elasticsearch 7.0 ships with a new cluster coordination subsystem that is faster, safer, and simpler to use. We heavily relied on **formal methods** to **validate our designs** up-front, with automated tooling providing strong guarantees in terms of correctness and safety. You can find the formal specifications of Elasticsearch's new cluster coordination algorithm in our **public Elasticsearch formal-models repository**. The core **safety module** of the algorithm is simple and concise and there is a direct one-to-one correspondence between the formal model and the **production code** in the Elasticsearch repository.

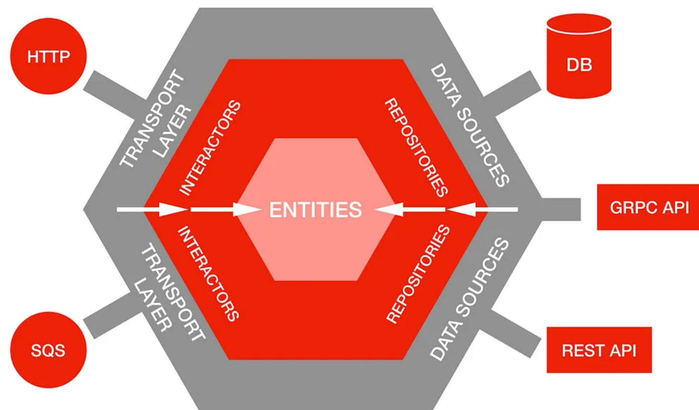
1





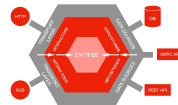
1. Elasticsearch bruger deres egen algoritme til at håndtere distribueret konsensus
2. Distribuerede systemer er svære at ræsonnere uformelt om
3. Model checking kan ikke skaleres til udtømmende beviser, og mange fejl sker kun i meget specifikke situationer
4. Safety: Hvad må aldrig ske
5. Liveliness: Hvad skal ske på et tidspunkt
6. Længere eksempel med Isabelle: `https://martin.kleppmann.com/2022/10/12/verifying-distributed-systems-isabelle.html`
7. Mere info om fejl i Elasticsearch: `https://aphyr.com/posts/323-jepsen-elasticsearch-1-5-0`

# Arkitektur







1. Formaliseret kerne: Hexagonal arkitektur
2. Idé: det er ligemeget for forretningslogikken hvad vores API er og hvor vi får data fra
3. Kun de svære dele bevises korrekte
4. Start i det små
5. Input/output kun i "kanterne"
6. Andre fordele: lettere at teste, lettere at udskifte API'er, databaser osv.

# Praktiske råd

1. Start i det små
2. Skær alt unødvendigt ud af modellen
3. Lav antagelser, og dokumentér dem
4. Husk at modeller aldrig er perfekte



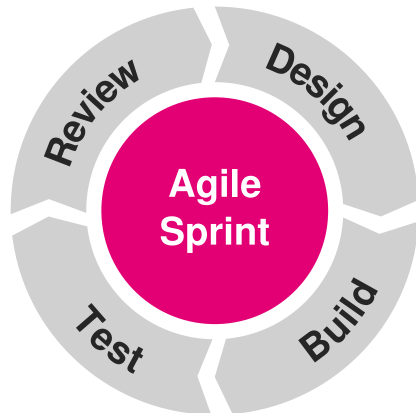


1. Start i det små
2. Skær alt unødvendigt ud af modellen
3. Lav antagelser, og dokumentér dem
4. Husk at modeller aldrig er perfekte

1. Specifikationer er aldrig perfekte
2. Lav modellen så lille som muligt (benhård fokus på det man laver lige nu)
3. Simplificer og lav antagelser
4. Fokus på hvad der skal gå godt og ikke hvad der kan gå galt
5. Hvor hurtigt kan det gå? Amazon: 2-3 uger, selv for begyndere
6. Inspiration: `https://assets.amazon.science/67/f9/92733d574c11ba1a11bd08bfb8ae/how-amazon-web-services-uses-formal-methods.pdf`

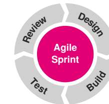
DTU

# Formelle metoder og agilitet



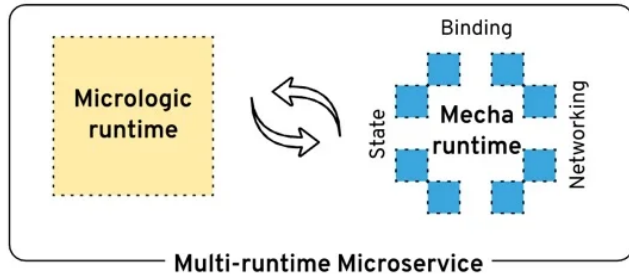




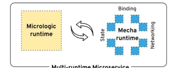

1. "Tænk på hvor meget tid og energi I allerede bruger på at løse problemer, der først opdages når systemet er i produktion. Sådanne problemer påvirker selvfølgelig brugeren eller kunden, men de påvirker også jeres udviklere. Det er stressende og demoraliserende at bruge tid på at slukke ildebrande." - Fra en bog om hvorfor man bør teste sin software
2. Mere mod = hurtigere udvikling i længden
3. Mere mod = mod til aggresive optimeringer
4. Mere mod = flere features
5. Mere præcision: lettere estimering af arbejdets størrelse ved ændringer
6. Agile har nogle gange et bias mod ting, der kan gøres hurtigt
7. Shift left: Giver kravspecifikationen egentlig mening?
8. Flere erfaringer: `https://link.springer.com/chapter/10.1007/978-3-030-57761-2_5`

# Fremtidsudsigter



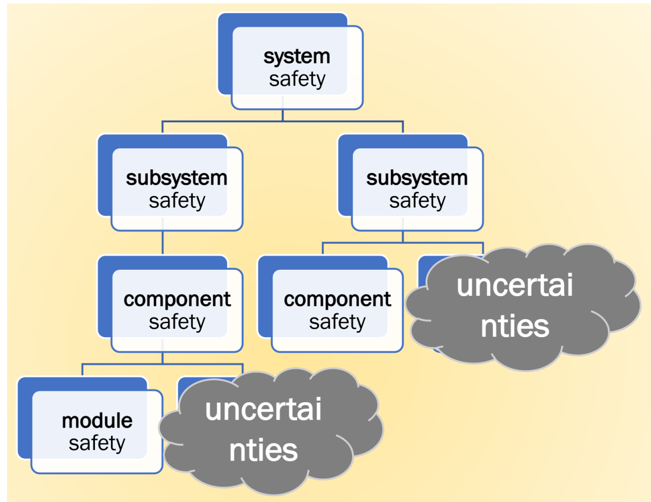




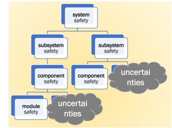
1. MECHA-arkitektur med formelt verificerede komponenter (perspektiv: formaliseret kerne)
2. Formelt verificerede biblioteker til svære ting (SSL, privacy osv.)
3. Eksempel: BoringSSL med kryptografi i Coq (`https://github.com/mit-plv/fiat-crypto`)

# Fremtidsudsigter



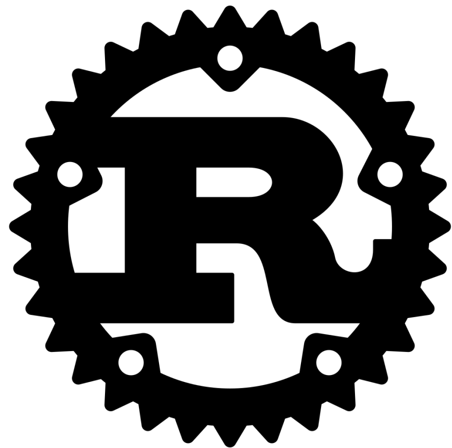




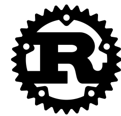
1. Delvis verificering med antagelser
2. Antagelser kan berettiges med domæneviden, f.eks. statistik eller fysik
3. Start fra toppen (hele systemet) i stedet for bunden

# Fremtidsudsigter









1. Stærkere typesystemer i almindelige sprog
2. Eksempel: Rust og affine typer

DTU

# Indgange til læring

- Information om Isabelle: `https://isabelle.systems/`
- Lærebøger med introduktion til Coq: `https://softwarefoundations.cis.upenn.edu/`
- Online introduktion til Lean: `https://adam.math.hhu.de/`





- Information om Isabelle: `https://isabelle.systems/`
- Lærebøger med introduktion til Coq: `https://softwarefoundations.cis.upenn.edu/`
- Online introduktion til Lean: `https://adam.math.hhu.de/`