# Kim jestem?

- Skromne 7 lat doświadczenia w przemyśle
- Automatyk z wykształcenia
- Projekty w systemach przeciwpożarowych, lotnictwie I HVAC (wbrew pozorom metody podobne)
- Entuzjasta metod MBD/MBSE
- Obecnie w Verum Software Tools BV

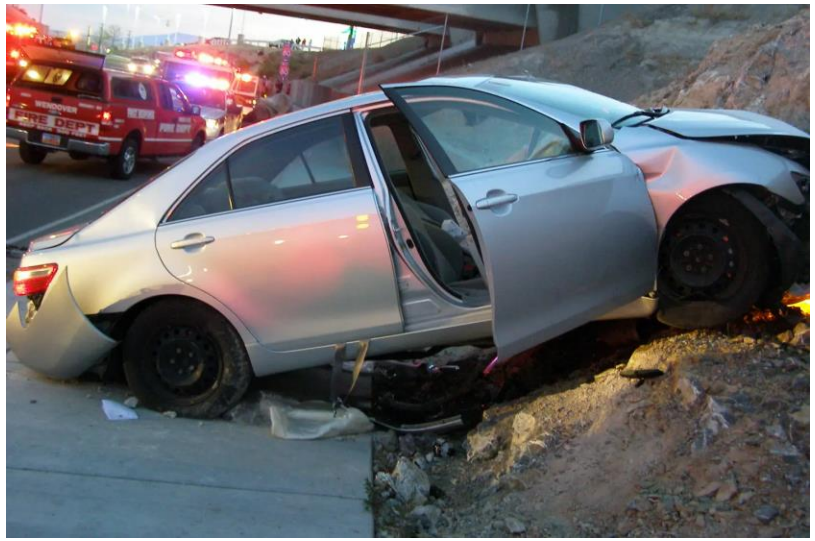
# Dlaczego tu jestem?

- Doświadczenia z miejsc pracy
- Chęć podzielenia się czymś fajnym

# Therac-25

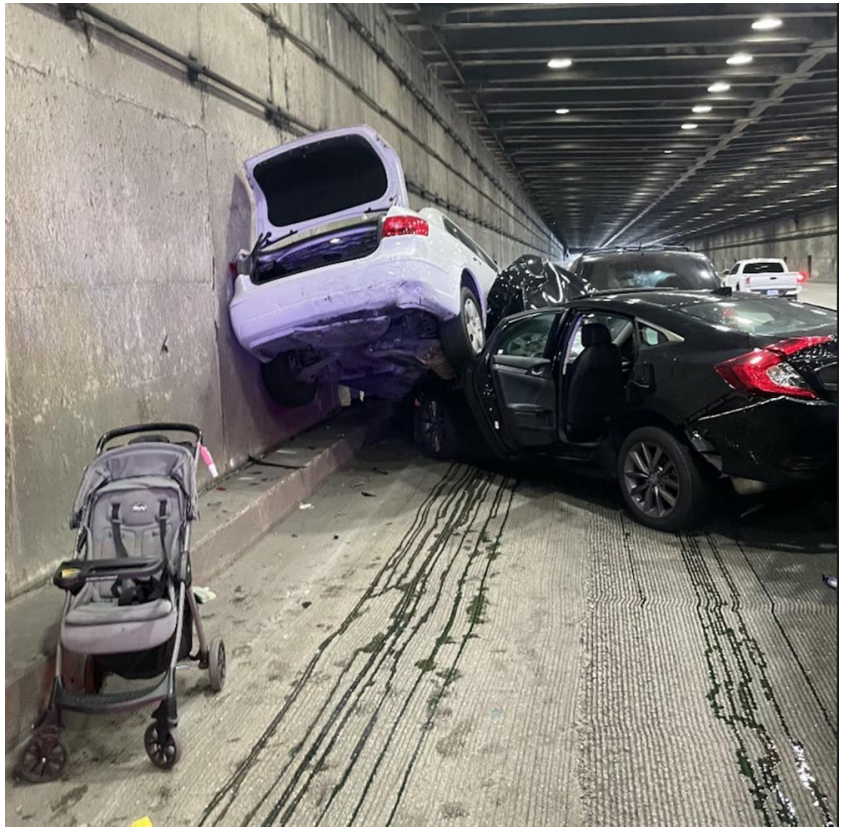
- Maszyna do radioterapi
- Przynajmniej 6 przypadków przedawkowania
- 3 zmarłych wskutek choroby popromiennej
- Błąd w kodzie sterującym – race condition

# Toyota: niekontrolowane przyśpieszanie



- Utrata kontroli nad mocą silnika
- Przynajmniej 89 przypadków śmietelnych
- Przynajmniej 57 osób rannych
- 1.2 miliarda $ odszkodowań
- Wiele błędów w kodzie, nieprawidłowe/niewystarczające procedury "fail safe"

# Tesla: wypadki spowodowane autopilotem



- 35 wypadków podczas których funkcja "Autopilot" najprawdopodbnie była aktywna
- 19 osób zabityvh
- "Phantom braking" jako jedna z możliwych przyczyn

# Znaczenie poprawności oprogramowania

**Gwałtownie rosnąca integracja systemów informatycznych i komunikacyjnych w różnych aplikacjach**

- Systemy wbudowane – pralka z 1MB programu to nic niezwykłego
- Protokoły komunikacyjne
- Środki transportu – samochody, samoloty, statki, pociągi itd.
- Automatyzacja, robotyzacja, autopiloty

**Błędy mogą skutkować ofiarami śmiertelnymi I ogromnymi stratami**

- Produkty masowej produkcji (Note 7, akcje serwisowe, piloty od telewizora)
- Systemy krytyczne dla bezpieczeństwa

# Czym jest weryfikacja systemu?

**Definicja:**
Weryfikacja systemu polega na sprawdzeniu, czy system spełnia określone wymagania jakościowe

**Weryfikacja != Walidacja**

- Weryfikacja:  "Czy tworzymy rzecz właściwie?"
- Walidacja: "Czy tworzymy właściwą rzecz?

# Techniki weryfikacji systemu

**Peer review**

- "Ręczna" inspekcja kodu bez jego uruchomienia
- Wykrywa od 31% do 93% błędów z medianą na poziomie 60%
- Słaba wykrywalność błędów w systemach współbieżnych I algorytmach (corner cases)

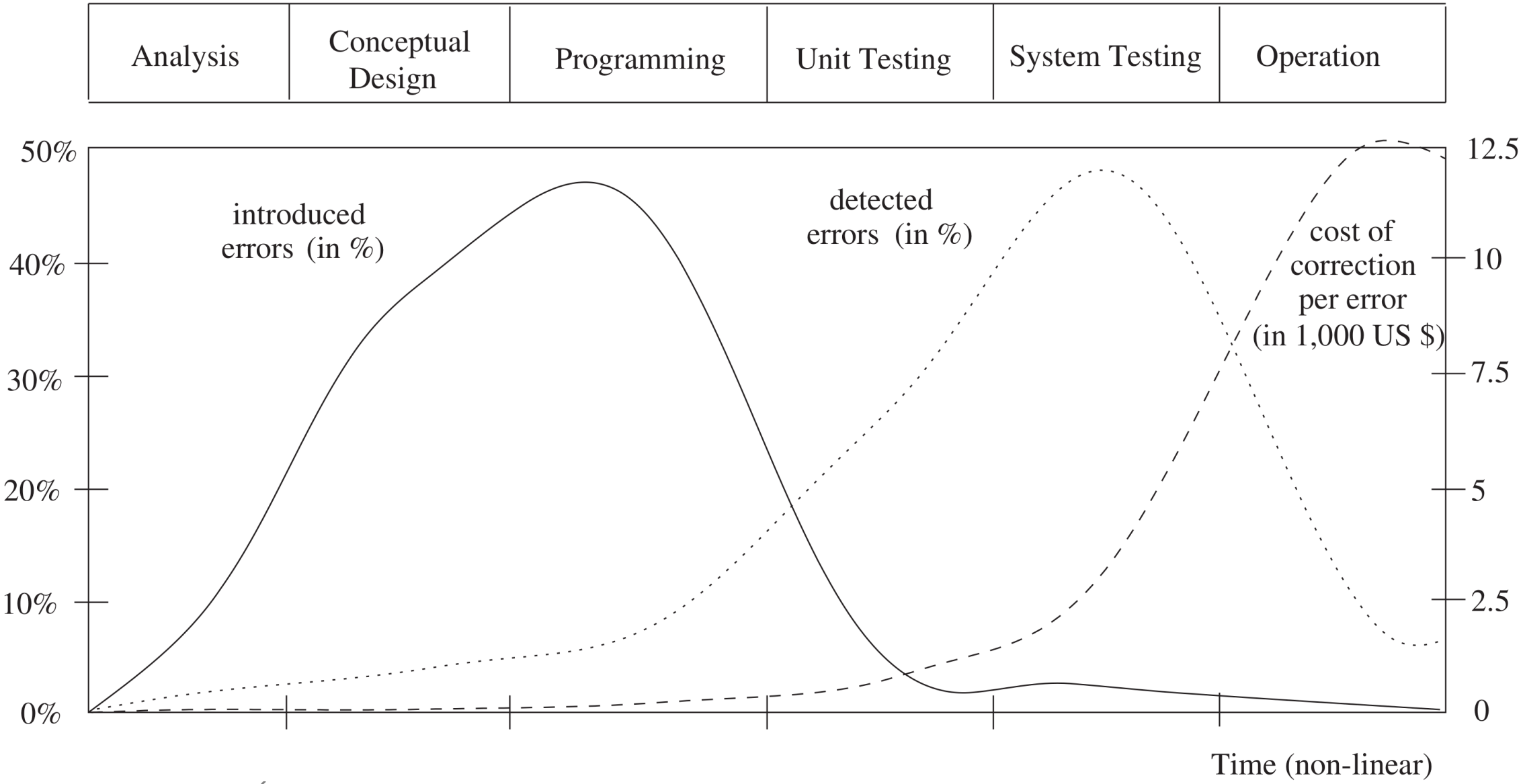
**Testy**

- Ręczne tworzenie przypadków testowych
- Generowanie danych testujących
- Techniki wymagające uruchomienia kodu

**Rezultaty**

- 30-50% kosztów projektu pochłaniają testy
- Testowanie średnio zajmuje więcej czasu niż programowanie
- Akceptowalne "zabugowanie" kodu to 1 błąd na 1000 linijek kodu (Microsoft)

# Szyciej wyłapiesz - mniej zapłacisz




Źródło: "Principles of Model Checking" Christel Baier, Joost-Pieter Katoen

# A jakby przetestować wszystko?

- Jak dużo mocy obliczeniowej potrzebuję?
- W jaki sposób wygenerować wszystkie przypadki testowe?
- W jaki sposób stwierdzić, że dany kod jest poprawny?

# Metody formalne

**Czym są?**

- Matematyka stosowana do modelewania I analizy systemów informatycznych I stosowanych
- Algorytmy dyskretne, struktury danych, teoria grafów, teoria automatów, logika itd.

**Co oferują?**

- Wczesna integracja weryfikacji w procesie projektu - już od pierwszej linijki kodu
- Lepszą efektywność oraz pokrycie kodu w porównaniu do tradycyjnych technik
- Skrócenie czasu weryfikacji

**Gdzie je używamy?**

- Krytyczne systemy bezpieczeństwa
- DO-178C(lotnictwo), ISO26262(automotive), PN-EN 61511(Bezpieczeństwo funkcjonalne)
- Rekomendowane przez FAA, IEC, NASA

# Rodzaje formalnej weryfikacji

**Metody dedukcyjne**

- Poprawność programu jako zbiór twierdzeń matematycznych zwanych warunkami weryfikacji
- Okreslenie warunków początkowych I pożądanych warunków końcowych programu
- theorem-proover/proof assistant
- Przydatne gdy system ma postać równań matematycznych

**Model checking**

- Weryfikacja KAŻDEGO stanu w jakim może znajdować się system
- mCRL2, SPIN, NuSMP, …
- Stosowany gdy system da się przetłumaczyć na skończony model behawioralny
- Stosuje techniki ograniczające poziom skomplikowania modelu(combinatorial explosion)

**Symulacje I testy w oparciu o modele(model-based)**

- Automatyczna generacja testów na podstawie wymagań

# Model checking

**Jak działa?**

- Zakłada, że system da się przedstawić w postacii maszyny stanów
- Sprawdza czy system spełnia określoną specyfikację
- Model I specyfikacja zostają sformułowane w języku matematycznym. Następnie kryteria specyfikacji zostają przedstawione jako formuła logiczna którą system spełnia lub nie spełnia

**Przykładowe kryteria specyfikacji**

- Deadlock/livelock (liveness properties)
- Determinizm
- Martwy/nieosiągalny kod
- Zgodność/spójność systemu – np. Czy w pełni implementujemy ten interfejs?
- Stany niedozwolone, poza zakresem I inne

# Przykłady z życia

**Microsoft**

- 85% crashy systemu spowdowane błędami w sterownikach od zewnętrznych dostawców
- Jeden z powodów - wysoki stopień skomplikowania API Windowsa
- SLAM - narzędzie do automatycznego sprawdzania czy dany sterownik jest zgodny z API

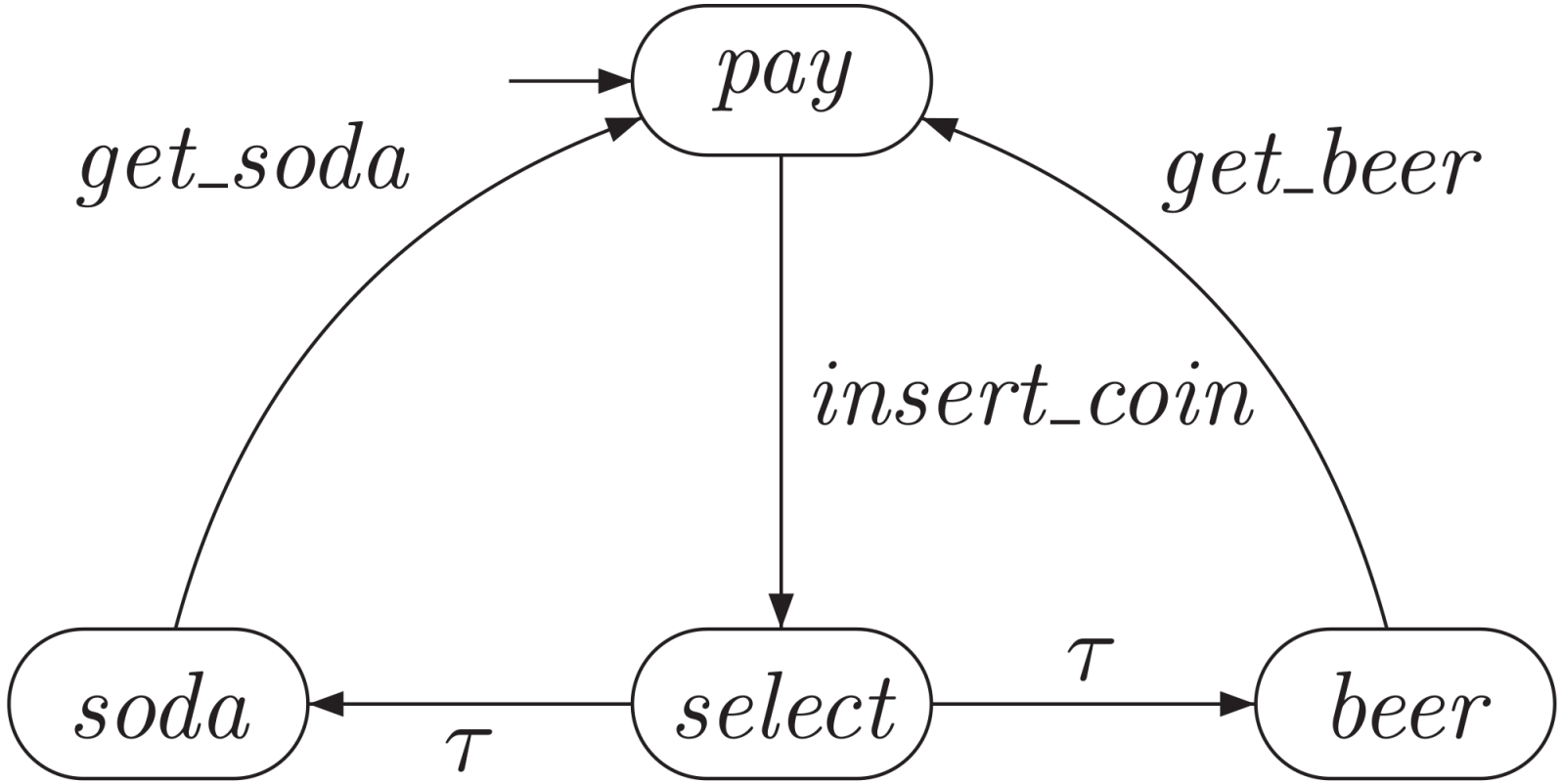
**Maeslantkering – ruchoma zapora wodna w Holandii**

- Czas otwierania/zamykania 5h
- Częstotliwość użycia średnio 1 raz na rok
- Zamknięta zapora to zablokowany port w Rotterdamie
- Sterowanie w całości przez program komputerowy

**Amazon, Facebook, Tesla, Nasa….**

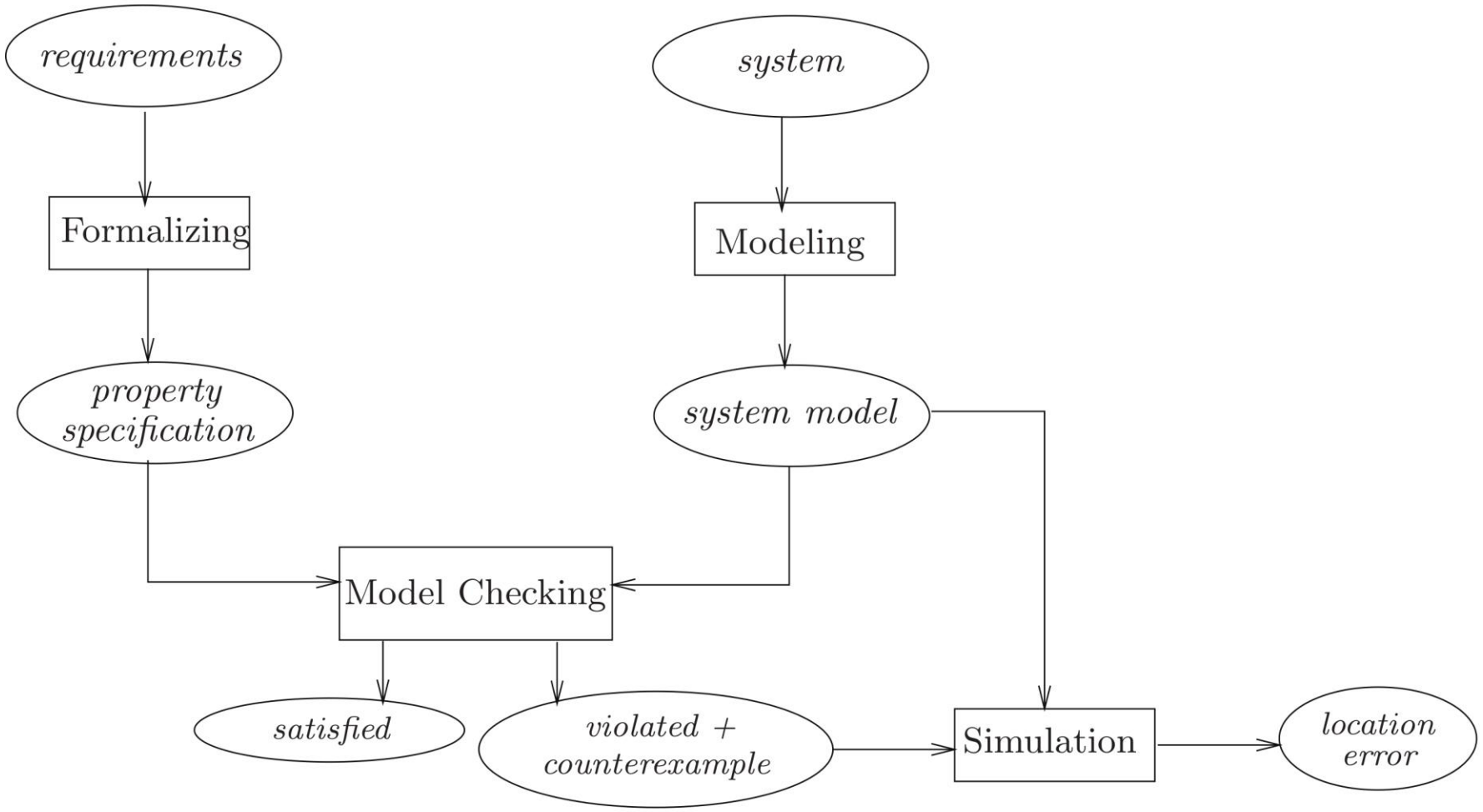
# Czym jest maszyna stanów?

**Maszyna stanów to nie tylko diagram Stateflow, switch z enumami, czy schemat w SysML. System w danej sekundzie pracy znajduje się w konkretnym stanie I muszą zajść pewne przejścia aby ten się zmienił !**




Źródło: "Principles of Model Checking" Christel Baier, Joost-Pieter Katoen

# Schemat działania




Źródło: "Principles of Model Checking" Christel Baier, Joost-Pieter Katoen

# Translacja miedzy modelami

Typowy schemat działania to przepisanie weryfikowanego modelu na język formalny, kompatybilny z odpowiednimi narzędziami (mCRL2, SPIN, NuSMV)

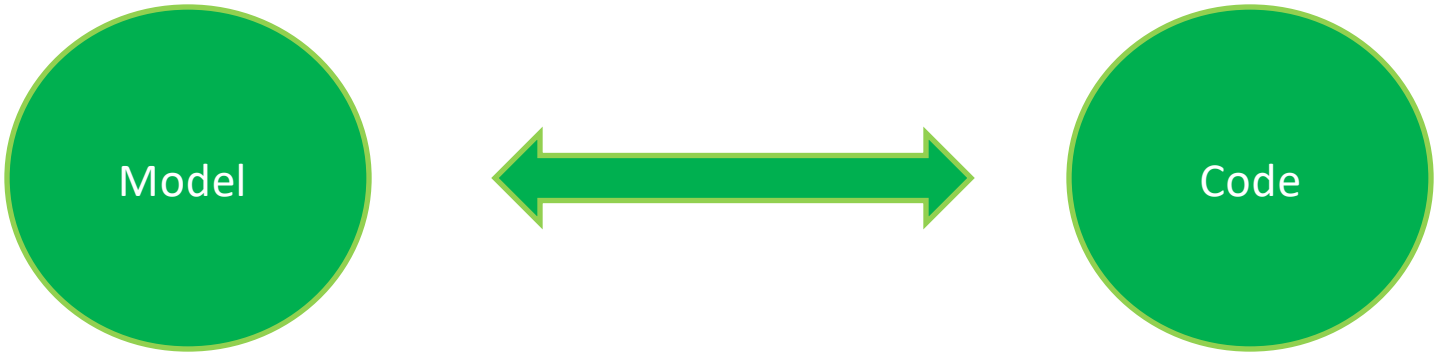
```
alarm'0(idle': Bool, sounding': Bool, grace': Bool) = alarm'1 ()
 + alarm'11 ()
 + alarm'16 ()
 + alarm'21 ()
 + alarm'34 ()
 + alarm'39 ();

alarm'1(idle': Bool, sounding': Bool, grace': Bool) = (idle') -> alarm'2 ();

alarm'2(idle': Bool, sounding': Bool, grace': Bool) = console'in (iconsole'action (icon

alarm'3(idle': Bool, sounding': Bool, grace': Bool) = alarm'4 ();

alarm'4(idle': Bool, sounding': Bool, grace': Bool) = auth'in (ipin'action (ipin'in'val
(valid'1)) . alarm'4_sum_helper(valid'1= valid'1);

alarm'4_sum_helper(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool)= alarm'5

alarm'5(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool) = (valid'1) -> alarm

alarm'6(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool) = alarm'7 ();

alarm'7(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool) = timer'in (itimer'a
(void)) . alarm'8 ();

alarm'8(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool) =  alarm'9(idle' = f

alarm'9(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool) = console'reply (ico

alarm'10(idle': Bool, sounding': Bool, grace': Bool, valid'1: Bool) = defer_skip(state_
alarm'behavior ();

alarm'11(idle': Bool, sounding': Bool, grace': Bool) = (!idle') -> alarm'12 ();

alarm'12(idle': Bool, sounding': Bool, grace': Bool) = console'in (iconsole'action (ico

alarm'13(idle': Bool, sounding': Bool, grace': Bool) = alarm'14 ();

alarm'14(idle': Bool, sounding': Bool, grace': Bool) = console'reply (iconsole'Bool(fal

alarm'15(idle': Bool, sounding': Bool, grace': Bool) = defer_skip(state_vector(idle',so

alarm'16(idle': Bool, sounding': Bool, grace': Bool) = (idle') -> alarm'17 ();
```

**Model mCRL2**

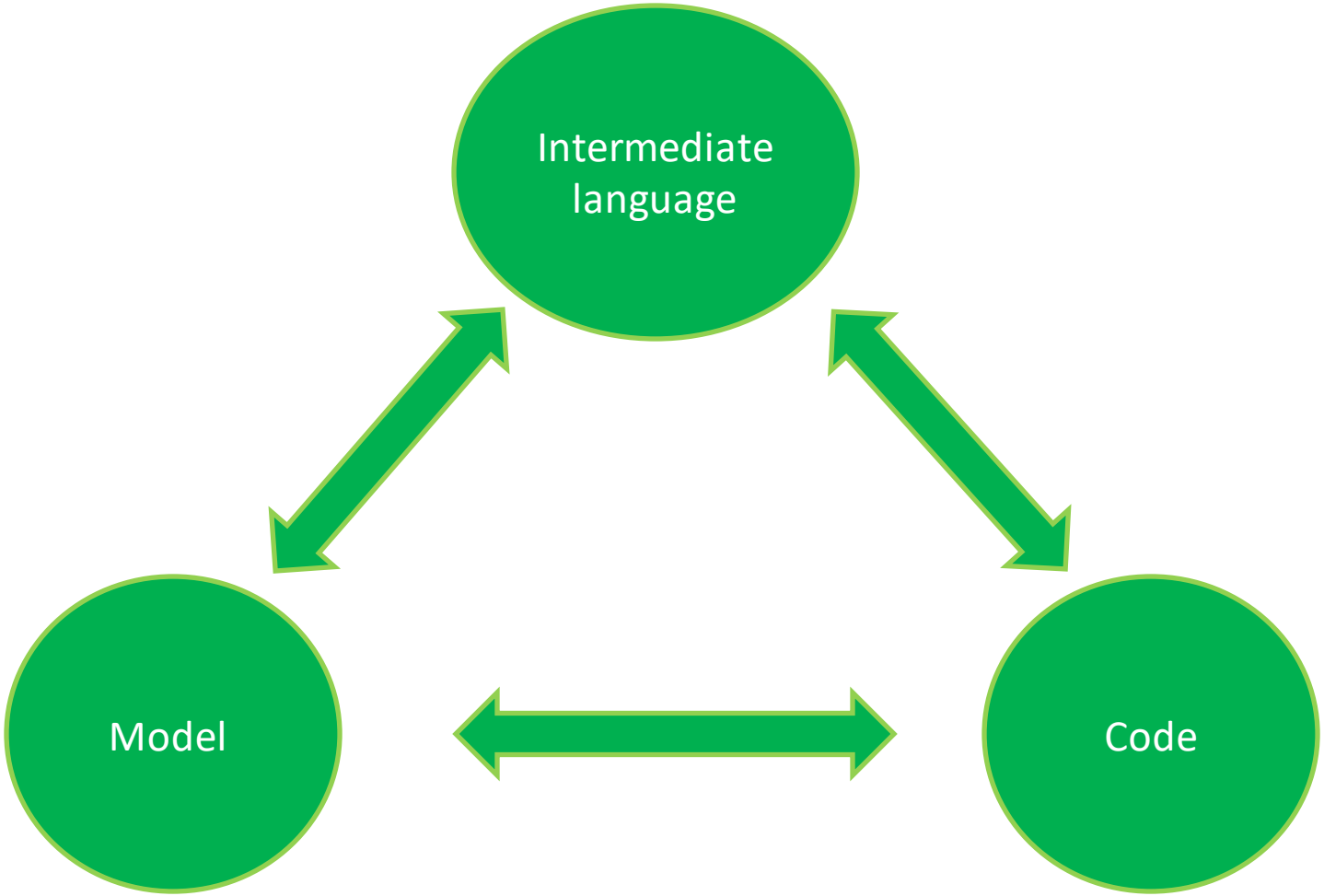
```
void alarm::console_arm(integer pin){
  if (idle)
  {
    bool valid = this->auth.in.valid(pin);
    {
      if (valid)
      {
        this->timer.in.set();
        idle = false;
      }
    }
    {
      this->reply_bool = valid;
      if (this->out_console) this->out_console();
      this->out_console = nullptr;
    }
  }
  else
  if (!idle) {
    this->reply_bool = false;
    if (this->out_console) this->out_console();
    this->out_console = nullptr;
  }
  else
  dzn_locator.get<dzn::illegal_handler>().illegal();
}
```

**kod C++**

# Schemat działania



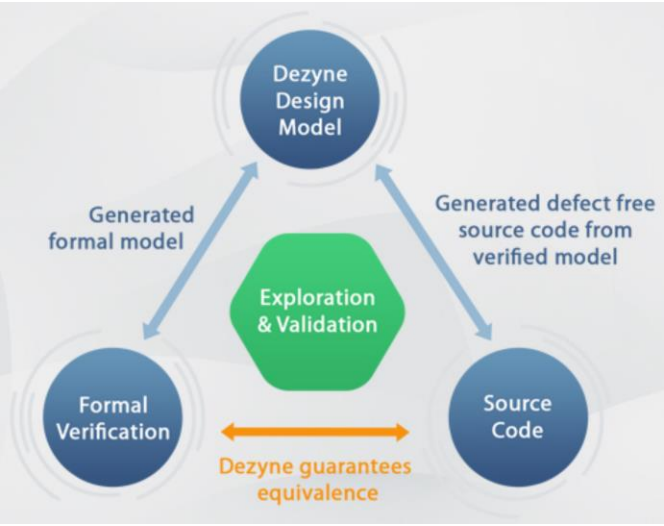
# Schemat działania



# Dezyne

**Czym jest Dezyne?**

- Otwarto-źródłowy język dziedzinowy
- Oparty o mCRL2
- Używany do projektowania struktur I zachowania systemu
- Składnia podobna do popularnych języków jak C/Java
- Używa podejścia "Design by contract" (interfejsy I komponenty)
- Struktura podobna jak w SysML
- Generowanie kodu na platofmę docelową

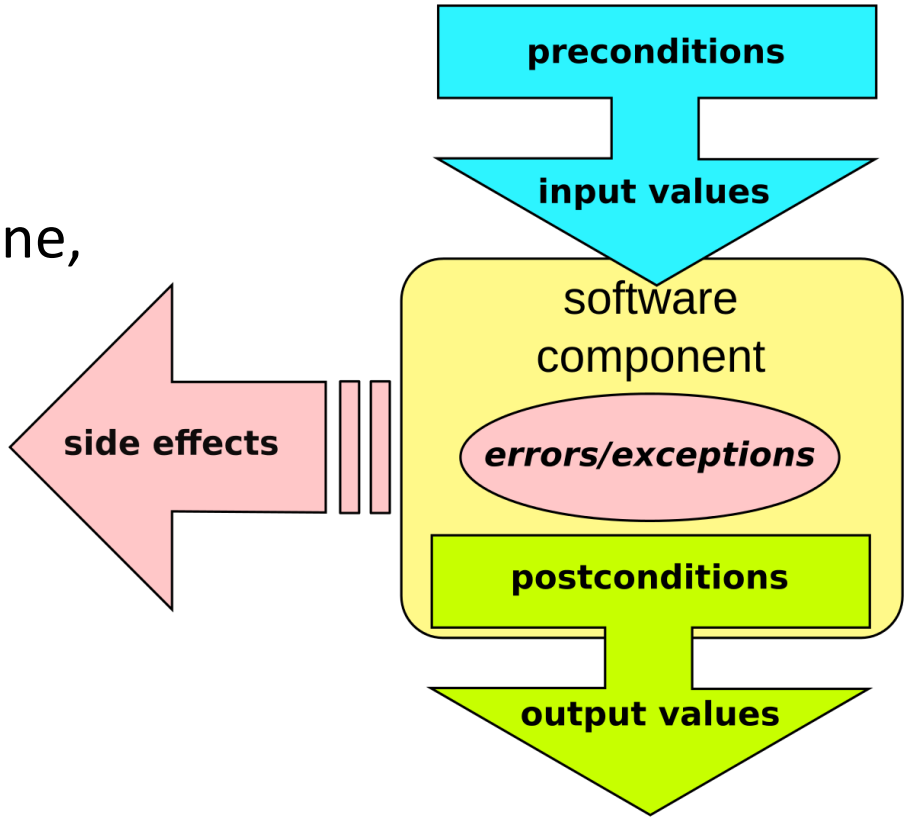



**Podobne rozwiązania**

- Verilog
- VC Formal (Synopsys)
- Simulink Design Verifier

# Design by contract

- Programowanie według umowy
- Narzuca definiowanie formalnych, precyzyjnych I weryfikowalnych
  specyfikacji interfejsów
- Rozrzeszają definisję typów danych o warunki wstępne, końcowe oraz
  niezmienniki

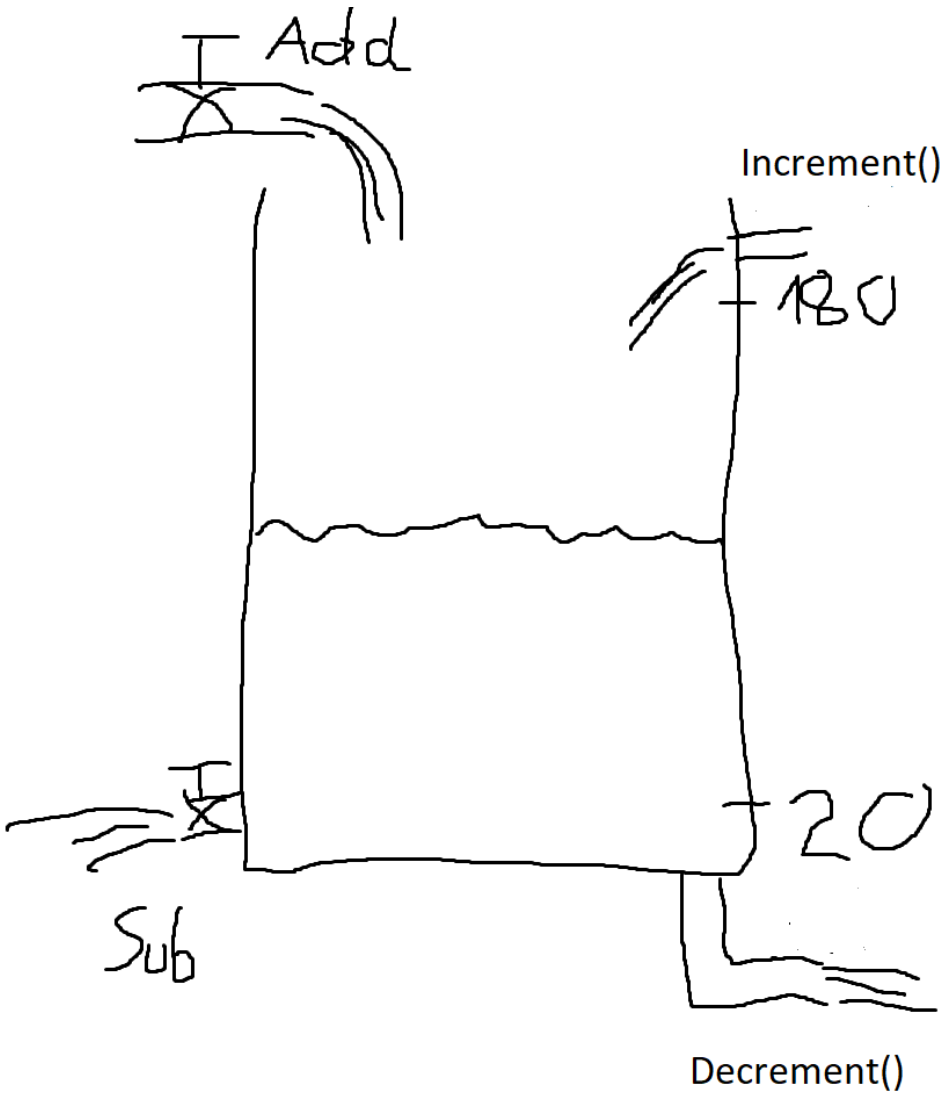


# Live coding – zbiornik wodociągów

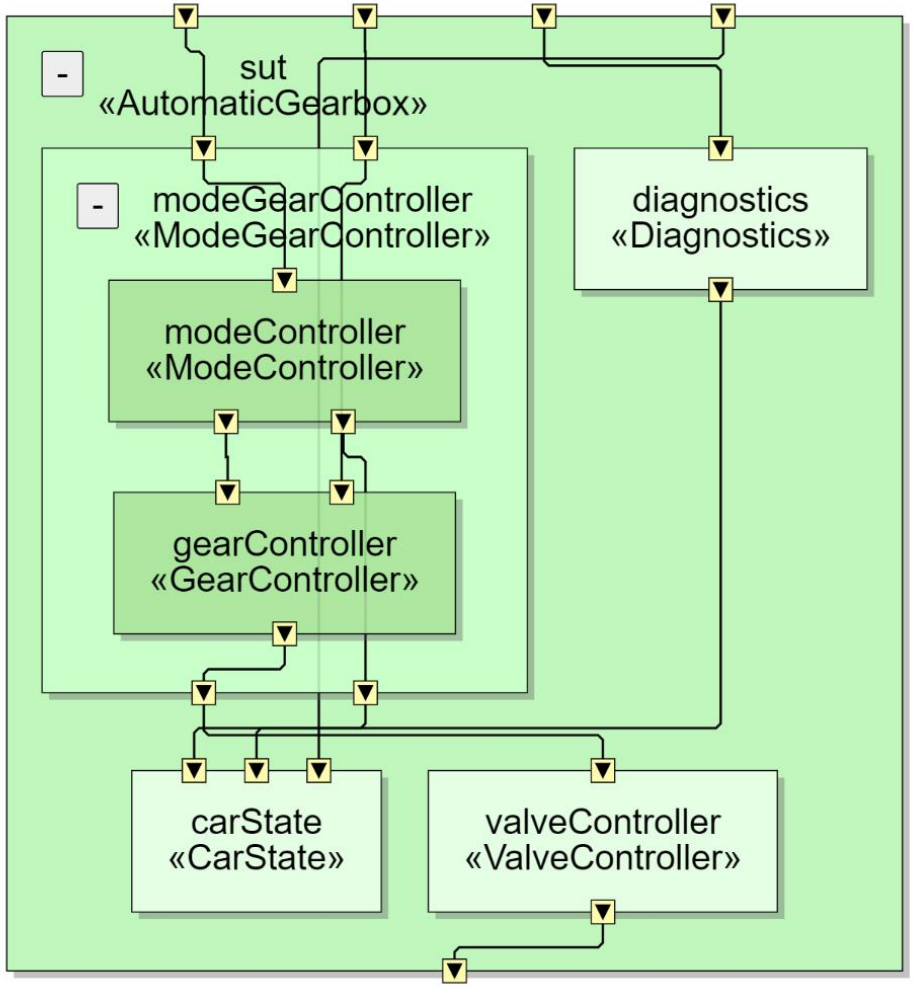
**Założenia:**

- Utrzymanie poziomu wody w zakresie 20-180
- Nieznany dopływ(Add)/odpływ wody(Sub)
- 2 zawory kontrolne mogące opróżniać (decrea

lub uzupełniać poziom wody (increase)



# Live coding - Skrzynia biegów

# Live coding – Generacja kodu

# Model checking – wady I zalety

**Zalety**

- Moze być szeroko stosowana(hardware, software, protokoły – tak, da się je przedstawić jako maszynę stanu)
- Pozwala na częściową weryfikację, cześci systemu(modułu, interfejsu)
- Weryfikacja za jednym przyciskiem guzika
- Szybko rosnące zainteresowanie ze strony przemysłu
- W przypadku wykrycia niezgodności, jest w stanie dostarczyć nam “steps-to-reproduce”
- Silne oparte na matematyce, brak dwuznaczności
- Bezstronność, możliwość sprawdzenia każdego przypadku

**Wady**

- Bardziej skupia się na logice sterowania niż na przetwarzanych danych
- Nie napisze za nas dobrych wymagań!
- Weryfikacja system jest tak dobra jak dobry jest model systemu
- Brak gwarancji kompletności – weryfikacja sprawdzi tylko te właściwości jakie określiliśmy
- Duże systemy mogą wymagać większych mocy obliczeniowych

# Źródła I kontakt

- “Principles of Model Checking”, Christel Bayer, Joost-Pieter Katoen
- “Advanced formal verification”, Rolf Dreschler
- http://savannah.nongnu.org/projects/dezyne


- Kontakt: karol.kobiela@verum.com
- Verum.com