

Proving Real-time Properties of Embedded Software Systems

Outline of Presentation

- Introduction and Preliminaries
- Previous Related Work
- Held_For Operator
- Sensor Lock System
- Verified Design for Timing Properties
- Delayed Trip System
- Optimization
- Conclusion

Real-time Safety Critical Systems

A system whose correctness depends on

- the system outputs values
- the times at which these outputs are generated

Failure results in:

- physical injury or loss of life
- unacceptable financial loss

Applications Areas:

- Medical equipment
- Aerospace
- Process control - e.g. Darlington Nuclear Generating Station
Shutdown Systems (SDS)

Motivation

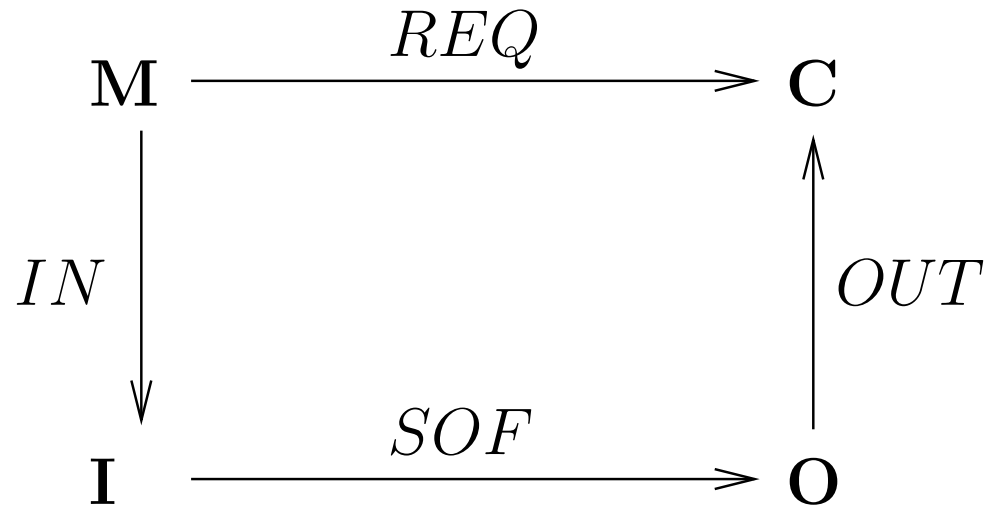
- Minor changes result in another extensive (&expensive) round of testing and review
- Capture and validate system requirements
- Guide the design and verification
- Reduce the verification work by modular design
- Approach to formally verified design optimization

Research Scope

- Will consider real-time systems in a discrete time setting
- Only one clock working in one real time system
- No concurrent clocks at different sample rates
- Ignore the intersample behaviour when modeling real-time systems
- No tolerance on timing specifications

Preliminaries

4-Variable Model (Parnas & Madey)



M - Monitored Variable statespace

C - Controlled Variable statespace

I - Input Variable statespace

O - Output Variable statespace

M, C, I, O are time series vectors and *REQ, SOF, IN, OUT* are relations.

We use a special case where all relations are functional resulting in proof obligation:

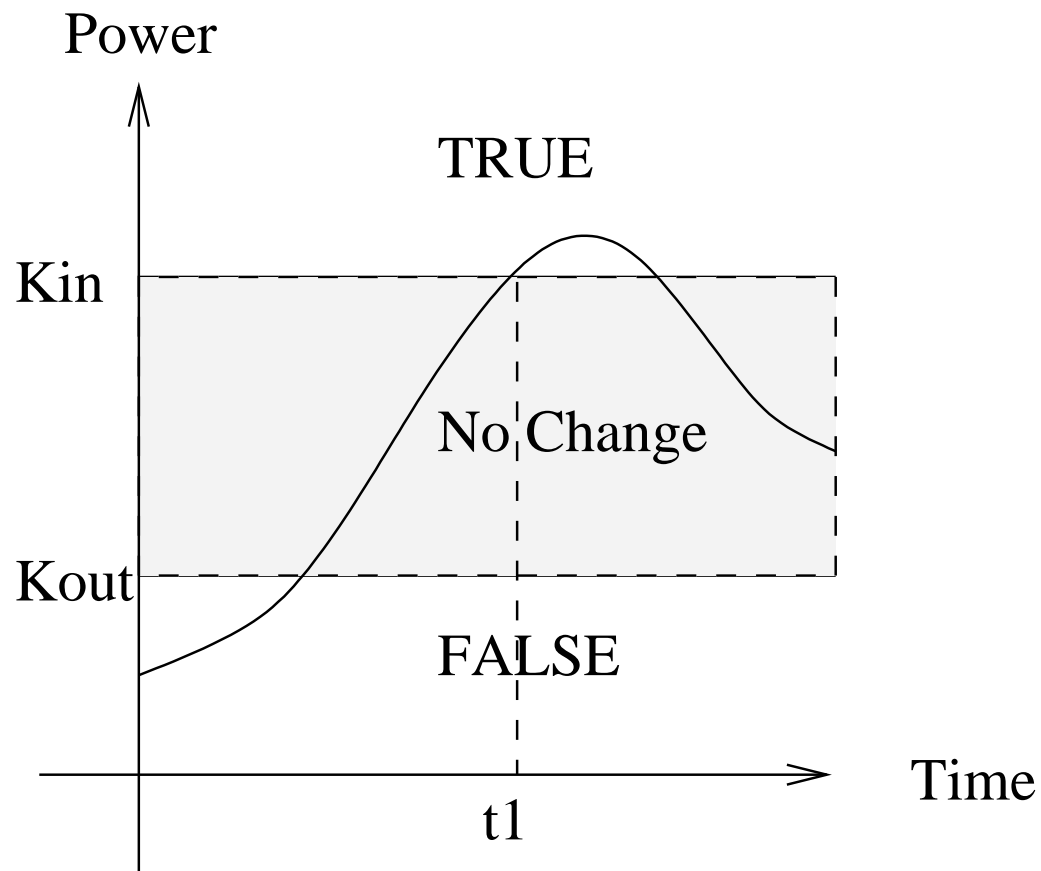
$$REQ = OUT \circ SOF \circ IN \quad (1)$$

Here *REQ* and *SOF* are the one step transition functions of the requirements and design respectively.

Example: Tabular Requirements

Many of the system requirements involve simple input/output logic, possibly dependent upon the previous value of the state variable:

e.g., Power Conditioning



$\text{PwrCond}(\text{Prev}:\text{bool}, \text{Power}, \text{Kin}, \text{Kout}:\text{posreal}):\text{bool} =$

| | | |
|---------------------------------|---|--------------------------------|
| $\text{Power} \leq \text{Kout}$ | $\text{Kout} < \text{Power} < \text{Kin}$ | $\text{Power} \geq \text{Kin}$ |
| <i>FALSE</i> | <i>Prev</i> | <i>TRUE</i> |

What about more complicated timing properties?

Clocks Theory (Dutertre and Stavridou)

For a positive real number T , we define a clock of period T , denoted $clock_T$, to be a set of “sample instances”

$$\begin{aligned} clock_T &:= \{t_0, t_1, t_2, \dots, t_n, \dots\} \\ &= \{0, T, 2T, \dots, nT, \dots\} \end{aligned}$$

For a period $T = 5$, the clock of period 5 is simply

$$clock_5 := \{0, 5, 10, 15, \dots\}$$

Note that $clock_5$, like all clocks as defined above, “starts” at time $t_0 = 0$.

Define $init$, $next_T$ and pre_T operators on the elements of $clock_T$ as follows:

$$\begin{aligned} \mathit{init}(t_n) &:= \begin{cases} \mathit{TRUE}, & n = 0 \\ \mathit{FALSE}, & \text{otherwise} \end{cases} \\ \mathit{pre}_T(t_n) &:= \begin{cases} t_{n-1}, & n \geq 1 \\ \text{undefined}, & \text{otherwise} \end{cases} \\ \mathit{next}_T(t_n) &:= t_{n+1} \end{aligned}$$

Held_For Operator (Lawford, Wu, OPG)

$$\text{pred}(\text{clock}_T) := \{f \mid f : \text{clock}_T \rightarrow \{TRUE, FALSE\}\}$$

$$\text{HELD_FOR} : \text{pred}(\text{clock}_T) \times \mathbb{R}^{\geq 0} \rightarrow \text{pred}(\text{clock}_T)$$

such that $(P)\text{HELD_FOR}(\text{duration})(t_n) = TRUE$ iff

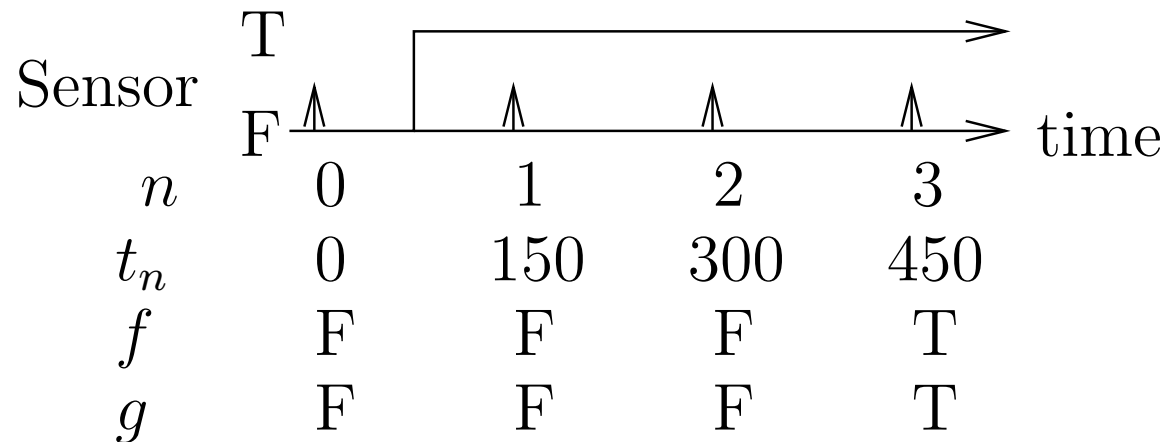
$$(\exists t_j \in \text{clock}_T)(t_n - t_j \geq \text{duration}) \wedge$$

$$(\forall t_i \in \text{clock}_T)(t_j \leq t_i \leq t_n \Rightarrow P(t_i))$$

Here we use $\mathbb{R}^{\geq 0}$ to denote non-negative real numbers.

Example:

Let $T=150$, $Sensor(t)$ be a clock predicate

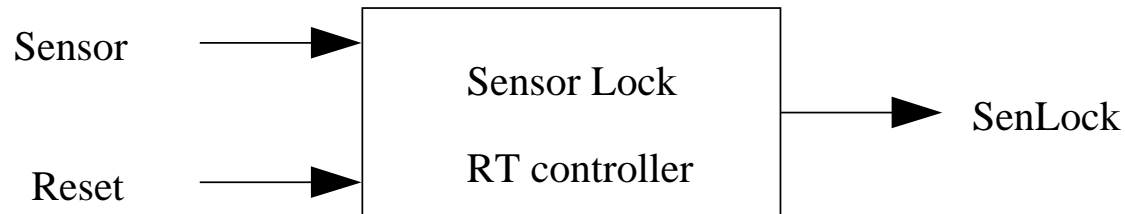


$$f(450) = (Sensor)Held_For(295)(450) = TRUE$$

$$g(450) = (Sensor)Held_For(310)(450) = FALSE$$

Note: We ignore the intersample behaviour of Sensor.

Sensor Lock System



Sensor Lock Real-time Controller

- Takes two boolean valued inputs, *Sensor* and *Reset*, and produces a single boolean valued output *SenLock* that is updated
- Sampling rate T , e.g., $T=100\text{ms}$

Behaviour

- When input *Sensor* is continuously *TRUE* for $k_ldelay = 150\text{ms}$ or longer, then the channel is “locked” and *SenLock* output is *TRUE*
- Once “locked” (i.e., *SenLock* becomes *TRUE*), the system will not “unlock” (*SenLock* becomes *FALSE*) until manually reset (*Reset = TRUE*)

Software Requirements

The required behaviour of the update function is summarized by the following table:

| <i>Condition</i> | | <i>Result</i> |
|---------------------------------------|--------------|---------------|
| | | SenLock |
| (Sensor) Held for (k_ldelay) | | TRUE |
| NOT [(Sensor) Held for (k_ldelay)] | Reset | FALSE |
| | \neg Reset | No Change |

Here $k_ldelay = 150\text{ms}$. When the conjunction of atomic proposition in a given row of the *Condition* columns is *TRUE*, then *SenLock* is set to the *Result* value for that row. E.g., when

$$NOT[(Sensor)Held_For(k_ldelay)] \wedge Reset$$

then $SenLock = False$.

Software Design

The SDD or “implementation” of this specification is given by the following table:

| | | <i>Results</i> | | |
|------------------|--------------------------------|----------------|-------|-------------|
| <i>Condition</i> | | Elock | LTime | |
| NOT Sensor | Elock | Reset | GOOD | 0 |
| | =LOCK | \neg Reset | LOCK | 0 |
| | Elock \neq LOCK | | GOOD | 0 |
| Sensor | LTime=0 | | BAD | next(LTime) |
| | $0 < \text{LTime} < k_ldelay$ | | NC | next(LTime) |
| | $\text{LTime} \geq k_ldelay$ | | LOCK | 0 |

Notes:

1. Here *Elock* has type $\{GOOD, BAD, LOCK\}$. The designer wants to use the additional information elsewhere in the system.

$$(Elock = LOCK) \equiv (SenLock = TRUE)$$

2. “NC” denotes “No Change”.
3. *LTime* is timer variable used to implement the *Held_For*.

Systematic Design Verification

SenLock_ELOCK: THEOREM

$$\text{SenLock}(t) = \text{LOCK?}(\text{Elock}(\text{ELOCK}(t)))$$

To apply PVS to this Verification Problem we use the strategy (INDUCT "t" 1 "clock_induction"). This breaks proof into two parts: (i) Base Case when $t=0$, and (ii) inductive case. In the course of proving these cases, we find the following errors:

1. Wrong initial condition for Elock.
2. Elock becomes unlocked without a manual reset.
3. Cases exist where manual reset unlocks the SenLock but not Elock.

Systematic Design Verification (cont)

The complete specification and design require fail-safe operation so the value of *SenLock* was initially set to *TRUE*. In the original design *Elock* was initialized to *BAD*.

The SDD becomes unlocked because the *LTime* counter is reset to 0 when *Elock* is set to *LOCK*. As a result the system loses the “history” of *Sensor*. Although *Elock* does not correctly implement this requirement as specified by *SenLock*, it also illustrates how *SenLock* could be made “safer”. When *Sensor* = *TRUE*, *Elock* will not allow a manual reset, while *SenLock* will permit such a reset if *Sensor* was *FALSE* in the recent past.

Systematic Design Verification (cont)

Taking these issues into consideration, we provide “fixed” versions of the specification and implementation below:

| <i>Condition</i> | | | <i>Result</i> |
|--------------------------------------|--------------|---------------|---------------|
| (Sensor) Held for (k_ldelay) | | | SenLock |
| NOT [(Sensor) Heldfor (k_ldelay)] | Reset | \neg Sensor | False |
| | | Sensor | No Change |
| | \neg Reset | | No Change |

| | | | <i>Results</i> | |
|------------------|-----------------------|-------------------|----------------|-------------|
| <i>Condition</i> | | | Elock | LTime |
| NOT Sensor | Elock =LOCK | Reset | GOOD | 0 |
| | | \neg Reset | LOCK | 0 |
| | Elock \neq LOCK | | GOOD | 0 |
| Sensor | LTime < k_ldelay | Elock \neq LOCK | BAD | next(LTime) |
| | | Elock=LOCK | LOCK | next(LTime) |
| | LTime \geq k_ldelay | | LOCK | NC |

Software Requirements Specification (SRS)

| <i>Condition</i> | | | <i>Result</i> |
|------------------------------------|--------------|---------------|---------------|
| (Sensor) Held_For (k_ldelay) | | | SenLock |
| (Sensor) Held_For (k_ldelay) | | | True |
| NOT [(Sensor) Held_For (k_ldelay)] | Reset | \neg Sensor | False |
| | | Sensor | No Change |
| | \neg Reset | | No Change |

Software Design Description (SDD)

| <i>Condition</i> | | | <i>Results</i> | |
|------------------|-----------------------|--------------|----------------|-------------|
| | | | Elock | LTime |
| NOT Sensor | Elock =LOCK | Reset | GOOD | 0 |
| | | \neg Reset | LOCK | 0 |
| | Elock \neq LOCK | | GOOD | 0 |
| Sensor | LTime < k_ldelay | | NC | next(LTime) |
| | LTime \geq k_ldelay | | LOCK | NC |

A Systematic Approach

Problem: Getting complicated timing properties right in the implementation can be difficult when designer has to start and stop timers to implement timing constructs.

Solution: Used pre-verified blocks of code to implement recurring types of timing requirements.

Timing Behaviour in SRS and SDD of Sensor Lock System

- SRS: $(Sensor)Held_For(k_delay)$
- SDD: $Sensor \wedge LTime \geq k_delay$

Why not reuse this verified design for Held_For operator?

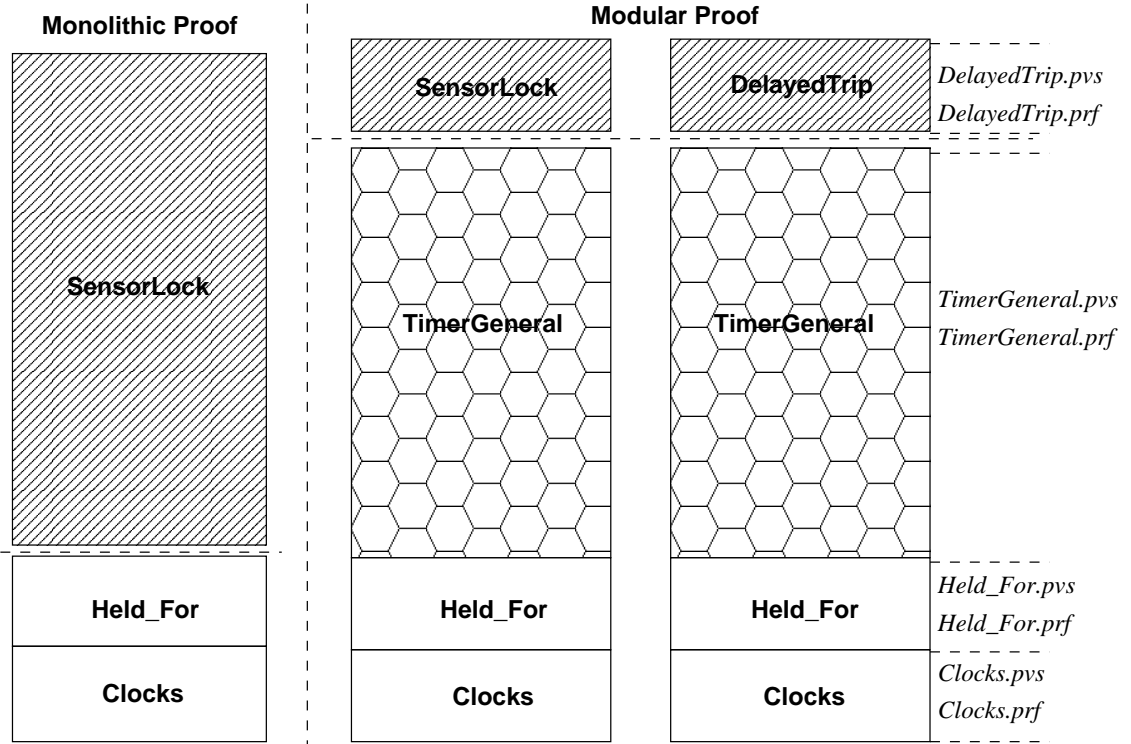
Monolithic Verification v.s. Modular Verification



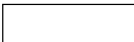
Monolithic Verification

- All-in-one structure for code and verification
- Difficult to identify and implement timing properties
- Difficult to verify the system
- Non-re-usability
- One change in requirement need to verify the whole system again.
- Potential performance advantage

Modular Verification

- Modularized structure easier for code maintenance
- Clear implementation of timing properties
- Easy to use and guarantee the correctness. Developer can easily use the module implementing a timing property
- Easier verification by instantiating a pre-verified theorem to complete the system specific verification
- Greatly reduced the verification work if changes happen to requirements
- Re-usability and Portability
- Overhead may exist due to modularization



-  **System Specific proof**
-  **Reusable Proof**
-  **Existing Proof**


```
TimerGeneral [T:posreal] : THEORY
```

```
BEGIN
```

```
  IMPORTING Held_For[T]
```

```
  t, previous: VAR clock
```

```
  P:var pred[clock]
```

```
  timeout : VAR posreal
```

```
  CurrentP:VAR bool
```

```
TimerUpdate(CurrentP,timeout,previous):clock= TABLE
```

```
          %------%
```

```
          | [previous<timeout|previous>=timeout] |
```

```
%------%
```

```
|CurrentP      | next(previous) | previous      ||
```

```
%------%
```

```
|NOT CurrentP  | 0                | 0            ||
```

```
%------%
```

```
ENDTABLE
```

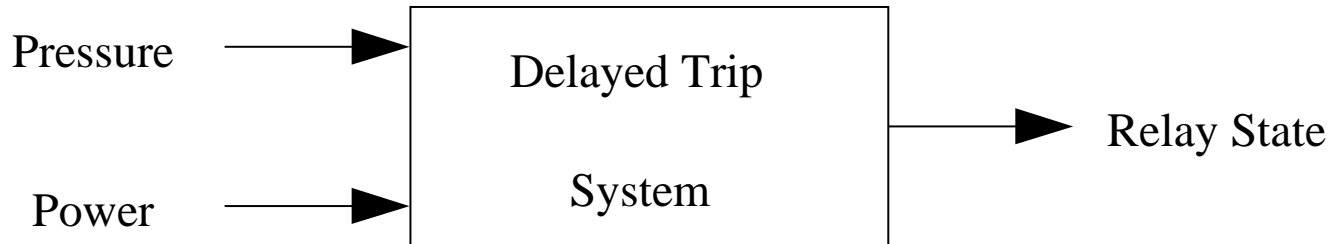
```
Timer(P,timeout)(t):RECURSIVE clock=  
IF init(t) THEN TimerUpdate(P(t),timeout,0)  
ELSE TimerUpdate(P(t),timeout,Timer(P,timeout)(pre(t)))  
ENDIF  
MEASURE rank(t)
```

TimerGeneral: THEOREM

```
IF init(t) THEN FALSE  
ELSE P(t) AND Timer(P,timeout)(pre(t))>=timeout ENDIF  
= Held_For(P,timeout)(t)
```

END TimerGeneral

Delayed Trip System



Delayed Trip Controller

- Takes 2 boolean inputs Power and Pressure; output Relay as time predicate
- If power and pressure both exceed the Power Threshold (PT) and Delayed Trip Point(DSP) for $k_timeout_1 = 3s$, then open the relay for $k_timeout_2 = 2s$. The DTS block's cycle time is $T = 100ms$, which means system inputting and updating every 0.1 seconds

Timing behaviour

- Need to use 2 Held_For Operators to specify requirements (SRS)
- Nesting of 2 Held_For Operators is necessary

Delayed Trip System SRS

| <i>Condition</i> | <i>Result</i> relay_open |
|---|-----------------------------|
| (PP) Held_For k_timeout1 | TRUE |
| $(\neg [(PP) \text{ Held_For } k_timeout1]) \text{ Held_For } k_timeout2$ | FALSE |
| $[\neg (PP) \text{ Held_For } k_timeout1] \wedge (\neg [\neg (PP) \text{ Held_For } k_timeout1]) \text{ Held_For } k_timeout2)$ | No Change |

Delayed Trip System PVS for Requirements

```
DelayedTrip_SRS(P,t,k_timeout1,k_timeout2): RECURSIVE bool=  
  IF init(t) THEN FALSE ELSE  
  LET NoChange = DelayedTrip_SRS(P,pre(t),k_timeout1,k_timeout2) IN  
  TABLE  
  |Held_For(P,k_timeout1)(t) | TRUE ||  
  |Held_For(NOT Held_For(P,k_timeout1),k_timeout2)(t) | FALSE ||  
  |NOT Held_For(P,k_timeout1)(t) &  
  (NOT Held_For(NOT Held_For(P,k_timeout1),k_timeout2)(t))| NoChange ||  
  ENDTABLE  
  ENDIF  
  MEASURE rank(t)
```

Delayed Trip System PVS for Design

```
SDD_State: TYPE = [# Relay: Relay_State, Timer1: clock, Timer2:clock #]  
  
RelayUpdate(k_timeout1,k_timeout2,CurrentP,S):Relay_State =  
  LET NoChange = Relay(s) IN  
  TABLE  
  |CurrentP & (Timer1(S) >= k_timeout1) | OPEN ||  
  |NOT (CurrentP&Timer1(S)>=k_timeout1) & Timer2(S)>=k_timeout2 |CLOSED ||  
  |NOT(CurrentP& Timer1(S)>=k_timeout1) & NOT( Timer2(S)>=k_timeout2)|NoChange||  
  ENDTABLE
```

Delayed Trip System PVS for Design (Continued)

```
SDD(Power,Pressure)(t):RECURSIVE SDD_State =
LET pp = Power(t)>=PT & Pressure(t)>=DSP IN
  IF init(t) THEN (#   Relay := CLOSED,
                    Timer1:=TimerUpdate(pp,k_timeout1,0),
                    Timer2:=TimerUpdate(TRUE,k_timeout2,0)   #)
  ELSE
  (#
    Relay:=RelayUpdate(Power(t),Pressure(t),SDD(Power,Pressure)(pre(t))),
    Timer1:=TimerUpdate(pp,
                        k_timeout1,
                        Timer1(SDD(Power,Pressure)(pre(t)))),
    Timer2:=TimerUpdate(NOT (pp & (Timer1(SDD(Power,Pressure)(pre(t))))>=floor(k_timeout1/T)*T)),
                        k_timeout2,
                        Timer2(SDD(Power,Pressure)(pre(t)))) #)
  ENDIF
MEASURE rank(t)

Timer1_Timer: lemma Timer(P,k_timeout1)(t)=Timer1(SDD(P,k_timeout1,k_timeout2)(t))

Timer2_Timer: lemma
Timer(NOT Held_For(P,k_timeout1),k_timeout2)(t)= Timer2(SDD(P,k_timeout1,k_timeout2)(t))

DelayedTrip_Block :THEOREM
DelayedTrip_SRS(PP,k_timeout1,k_timeout2)(t)= OPEN?(Relay(SDD(PP,k_timeout1,k_timeout2)(t)))
```

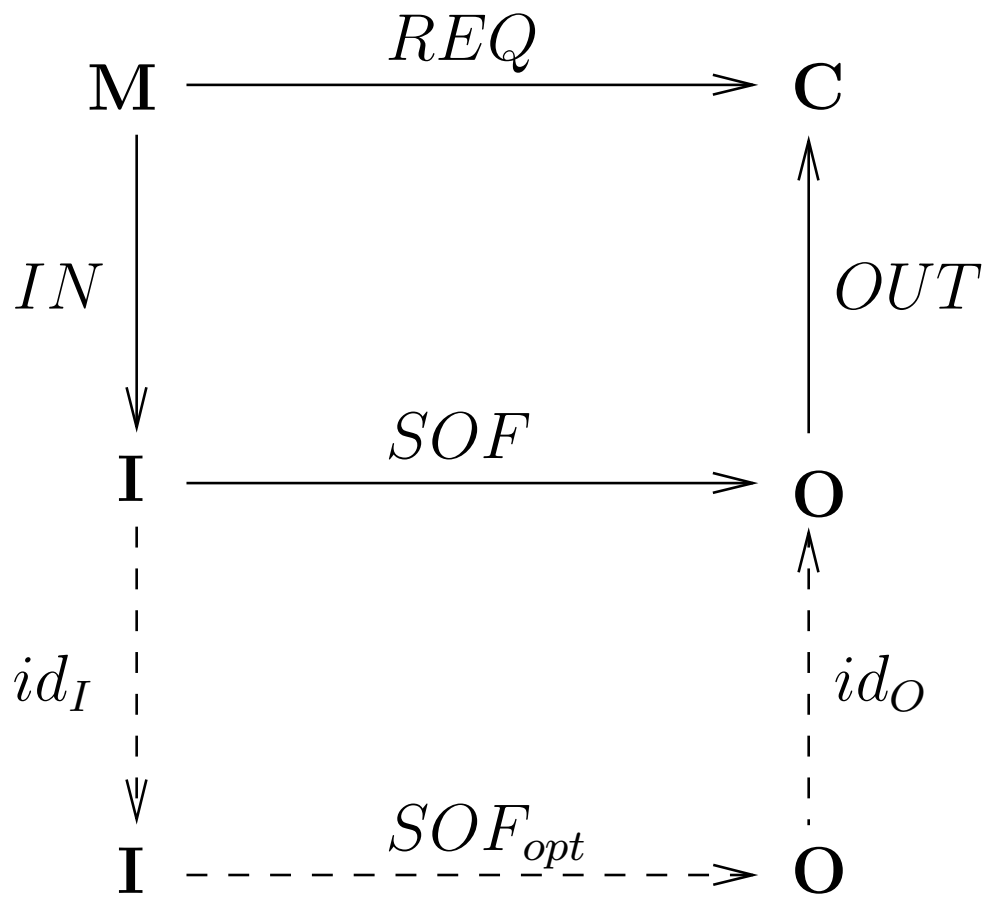
Performance and Verification

Objective:

- Improve the performance
- Guarantee the correctness of Optimized Code (Implementation)
- Save the verifier and developer from repeated work

Methods:

- Use PVS theorem prover's rewriting and propositional simplification
- Extend the Systematic Design Verification approach



DTS fastSDD implementation

How do you produce the “optimized” version with fewer conditional evaluations?

Define function constant `fastSDD` of same type as `SDD`, then try to prove they are equivalent.

```
fastSDD(Power,Pressure)(t):SDD_State
```

Optimize: PROPOSITION

```
fastSDD(Power,Pressure)(t)= SDD(Power,Pressure)(t)
```

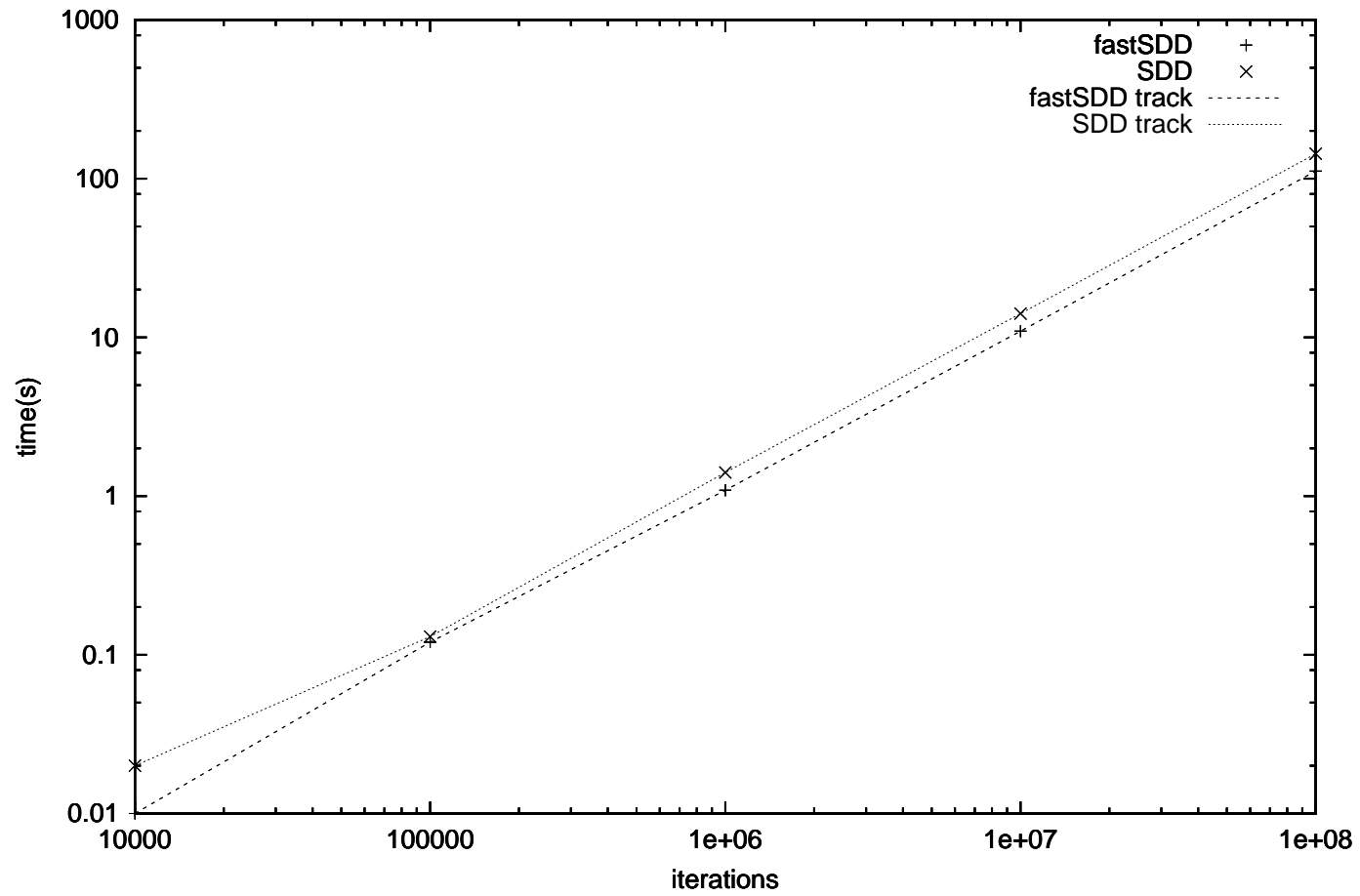
You can actually use PVS to produce the optimized code by (`skolem!`), rewrite the definitions of `SDD` and `TimerUpdate`, then (`bddsimp`) and interpret the unprovable sequents as distinct case.

There are now some compilers that do this sort of thing for you.

Pseudo code for “Optimized” Delayed Reactor Trip Example

```
IF Power>=PT & Pressure(t)>=DSP THEN
  IF Timer1>=k_timeout1 THEN
    Relay := OPEN,
    Timer2 := 0
  ELSIF Timer2>=k_timeout2 THEN
    Timer1 := Timer1 + T
  ELSE
    Timer1 := Timer1 + T,
    Timer2 := Timer2 + T
  ENDIF
ELSIF Timer2>=k_timeout2 THEN
  IF Power(t)<PT THEN
    Relay := CLOSED,
    Timer1 := 0
  ELSE
    Timer1 := 0
  ENDIF
ELSE
  Timer1 := 0,
  Timer2 := Timer2 + T
ENDIF
```

Performance Evaluation



Conclusion

- PVS-RT method deliver a guarantee of domain coverage.
- Pre-verified timing properties provide significant aid to PVS-RT method for re-usability and portability
- Formally verified Design Optimization
- Also PVS theorem prover can help to validate the requirements

Future Work

- Held_For operator with timing tolerance
- Consider applying different clock rates for data streams in the real-time systems.
- Create a real-time property verification library, including different timing operators.