From Stateflow Simulation to Verified Implementation: A Verification Approach and A Real-Time Train Controller Design

Yu Jiang[2], Yixiao Yang[1], Han Liu[1], Hui Kong[1], Ming Gu[1], Jiaguang Sun[1], Lui Sha[2]

[1] Tsinghua University

[2] University of Illinois at Urbana-Champaign

Outline

- **Motivation**
  - Enhanced verification for Stateflow
- **Background**
  - Stafteflow, Timed automata, runtime verification
- **Approach**
  - Stateflow to timed automata,
  - Runtime verification customization
- **Experiments**
  - Real train control system design
- **Conclusion**

Outline

- **Motivation**
  - Enhanced verification for Stateflow
- **Background**
- **Approach**
- **Experiments**
- **Conclusion**

Motivation-Worldwide Used Stateflow

- ***Simulink*** is widely used for model driven development, which provides delicate support for graphical ***Stateflow*** modeling, interactive model level simulation, some basic design validation, along with C, C++, and VHDL code generation and verification 。

- Simulink ***Design Verifier*** and Simulink ***Polyspace*** are taking the responsibility to uncover design defects and implementation defects, respectively.

Motivation-Stateflow Verification Limitation

- **Simulink Design Verifier –Design Defects**

  The verification capability of Simulink Design Verifier is limited to basic properties such as division by zero. *Handling complex temporal properties (e.g. something has to hold at the next state) of those applications is currently infeasible because of the limited descriptive ability of Simulink verification block.*

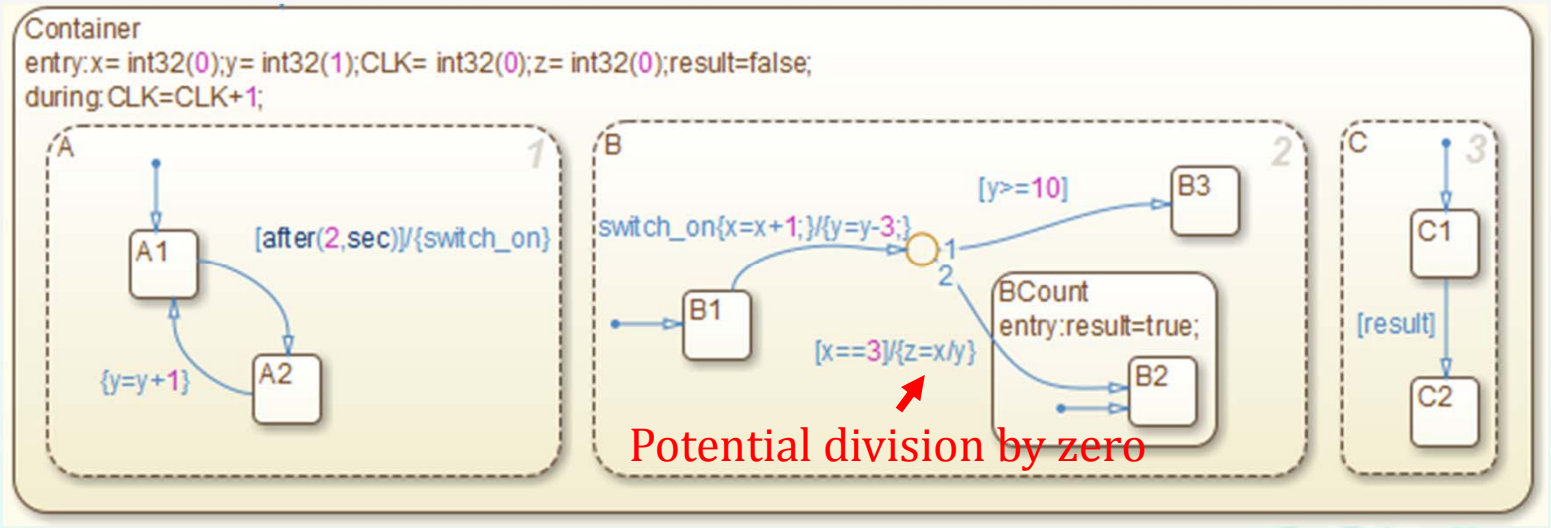
- **Simulink Polyspace – Implementation Defects**

  Simulink Polyspace offers the flexibility to check correctness over the implementation code using abstract interpretation techniques, *we still lack the knowledge to analyze the interaction between the target software and dynamic physical execution environment.*

Enhanced verifications should be considered.

Motivation- Verification Limitation Example

- When the Stateflow model enters the composite state B, there is a potential error of division by 0 contained in the transitional action z = x=y.



- Verify the property non-division by zero in Design Verifier, but the model passes the verification.

How to correct the mistake?

Motivation-Enhanced Verification Approach

- ## Enhancement of Simulink Design Verifier

  Supporting tools with stronger verification power such as Uppaal is expected here to check the properties of Stateflow model.

- ## Enhancement of Simulink Polyspace

  More rigorous formal techniques such as runtime verification should be applied to monitor and ensure the correctness of the automatically implemented systems.

Main Challenge

- The execution semantics of Stateflow is too complex, which is described in a 1366 pages user guide informally, which is non straight forward to formalize for verification.

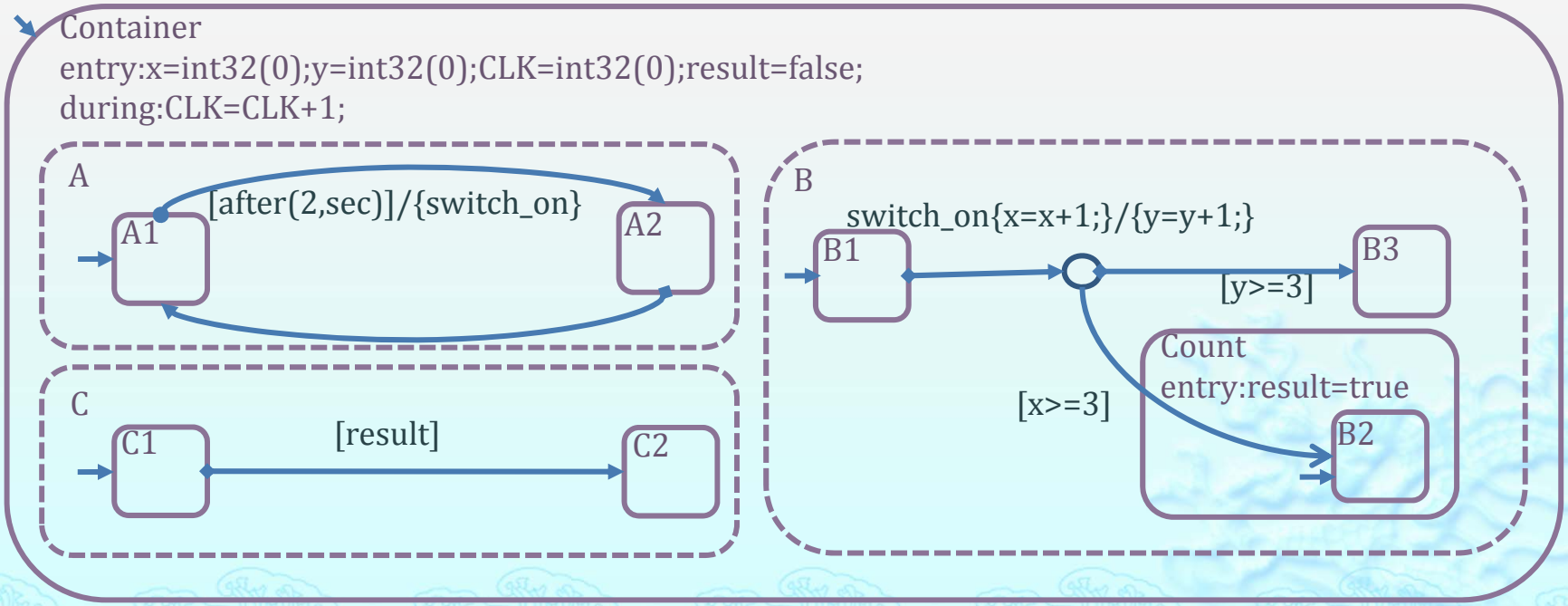
- The temporal and consistency verification of properties on the generated code (VHDL and C) running in an unexpected dynamic physical environment is hard to address, which is essential for safety critical applications.

Outline

- **Motivation**
- **Background**
  - Stafteflow, Timed automata, Runtime verification
- **Approach**
- **Experiments**
- **Conclusion**

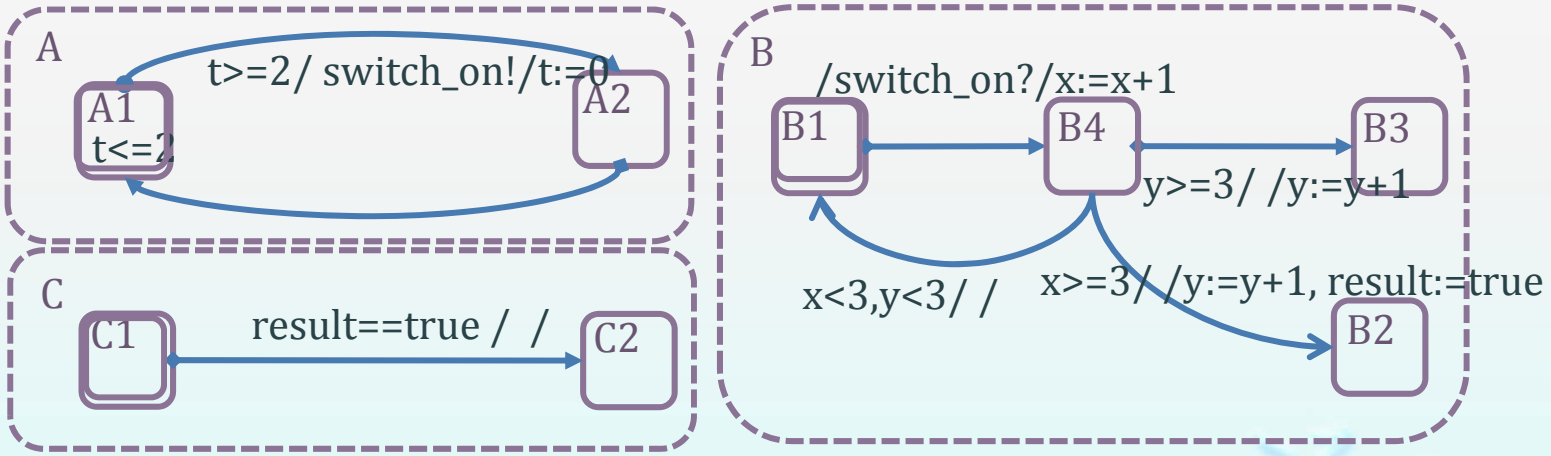
Background-Simulink Stateflow

- An example of a Stateflow diagram which covers most advanced modeling features. There are mainly six frequently-used modeling elements: *State, Transition, Junction, event, Action and Timer.*



Background-Timed automata

- An example of a timed automata which covers most advanced modeling features.



Background-Runtime verification

- Runtime verification is a lightweight verification technique by working directly with the actual system, thus scaling up relatively well and giving more confidence.

```
public class HasNext_1 {
    public static void main(String[] args){
        Vector<Integer> v = new Vector<Integer>();

        v.add(1);
        v.add(2);
        v.add(4);
        v.add(8);

        Iterator i = v.iterator();
        int sum = 0;

        sum += (Integer)i.next();
        sum += (Integer)i.next();
        sum += (Integer)i.next();
```

code

Event Definition

```
event hasnext after(Iterator i) :
    call(* Iterator.hasNext()) && target(i) {}
event next before(Iterator i) :
    call(* Iterator.next()) && target(i) {}
```

Property Definition

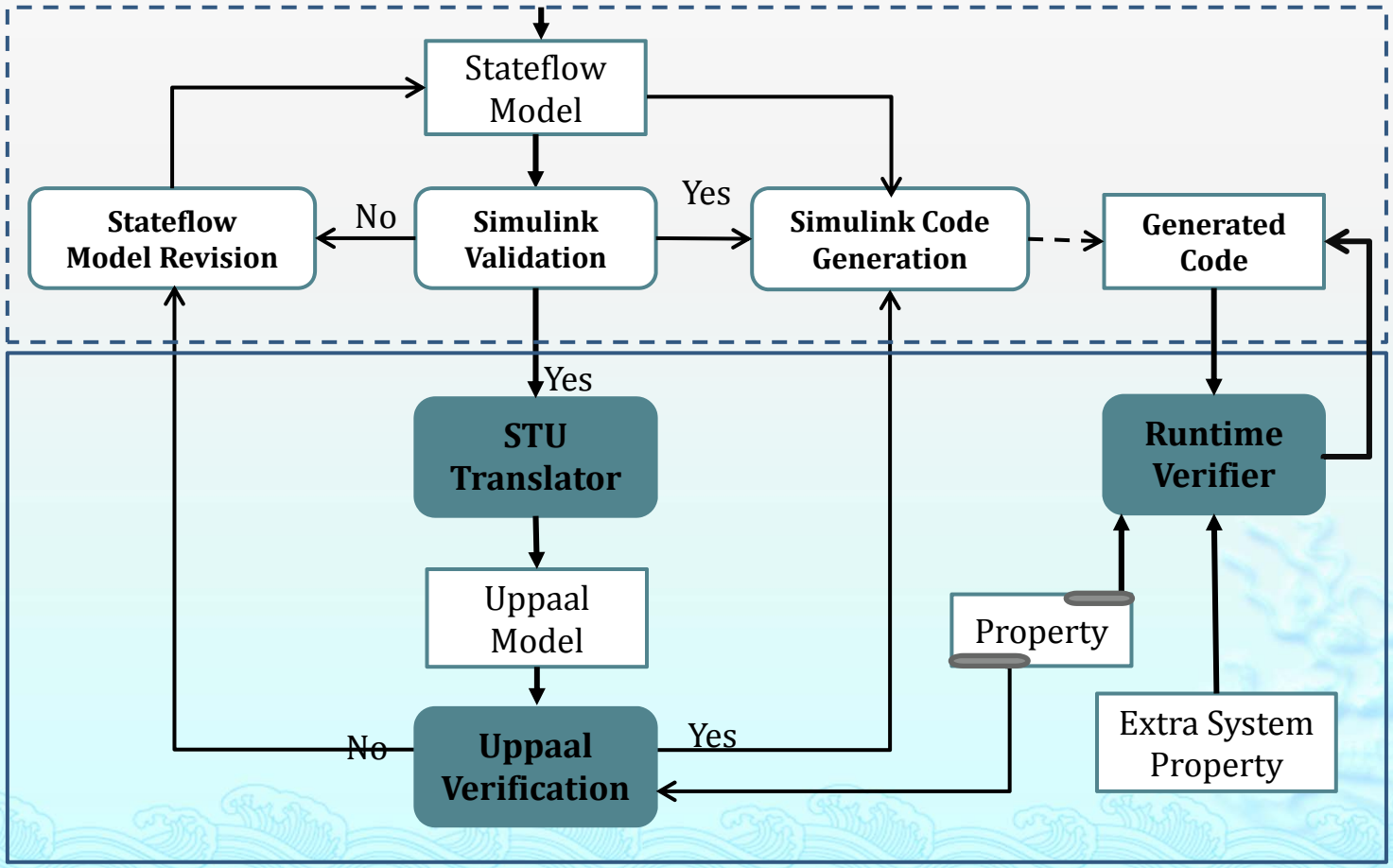
```
fsm :
  start [
     next -> unsafe
     hasnext -> safe
  ]
  safe [
     next -> start
     hasnext -> safe
  ]
  unsafe [
     next -> unsafe
     hasnext -> safe
```

Outline

- **Motivation**
- **Background**
- **Approach**
  - Stateflow to timed automata,
  - Runtime verification customization
- **Experiments**
- **Conclusion**

Approach-Enhanced Verification Overview

- The approach for the enhanced verification is presented as below.



Stateflow to Timed automata-Challenges

- The most challenging task is to overcome the semantics gap :
  - Stateflow transition is driven by event. Execution of every event is in deterministic sequential order, and interruptible with stack. *While timed automata is executed in parallel, and driven by the channel synchronization without the support of stack.*
  - Stateflow supports hierarchy structure which is combined with recursive activation-deactivation mechanism, transitional action, and conditional action very closely. *While timed automata support single state.*

Solve the gap by a virtual event stack, with complex operations on the stack.

Stateflow to Timed automata-Event Stack

- The key idea to simulate Stateflow event stack mechanism is to build a virtual stack in Uppaal. We use a structured array in Uppaal to build the event virtual stack.

```
Structure Event {
  int  Event;
  int  Dest;
  int  DestCrossPosition;
  int  AutomatonType;
  bool Valid;
}
```

- We also provide five basic functions DispatchEvent(), PushEvent(), PopEvent(), EventSentToMe(), and StackTopEvent() for Uppaal to accomplish interrupt and recursive activation-deactivation mechanism。

Stateflow to Timed automata- Stack based Deactivation

```
Void StateDeactivationLogic(int state_s^f)
{
   DispatchDeactivationToChild(state_s^f);
   HandleDeactivation(state_s^f);
}
```

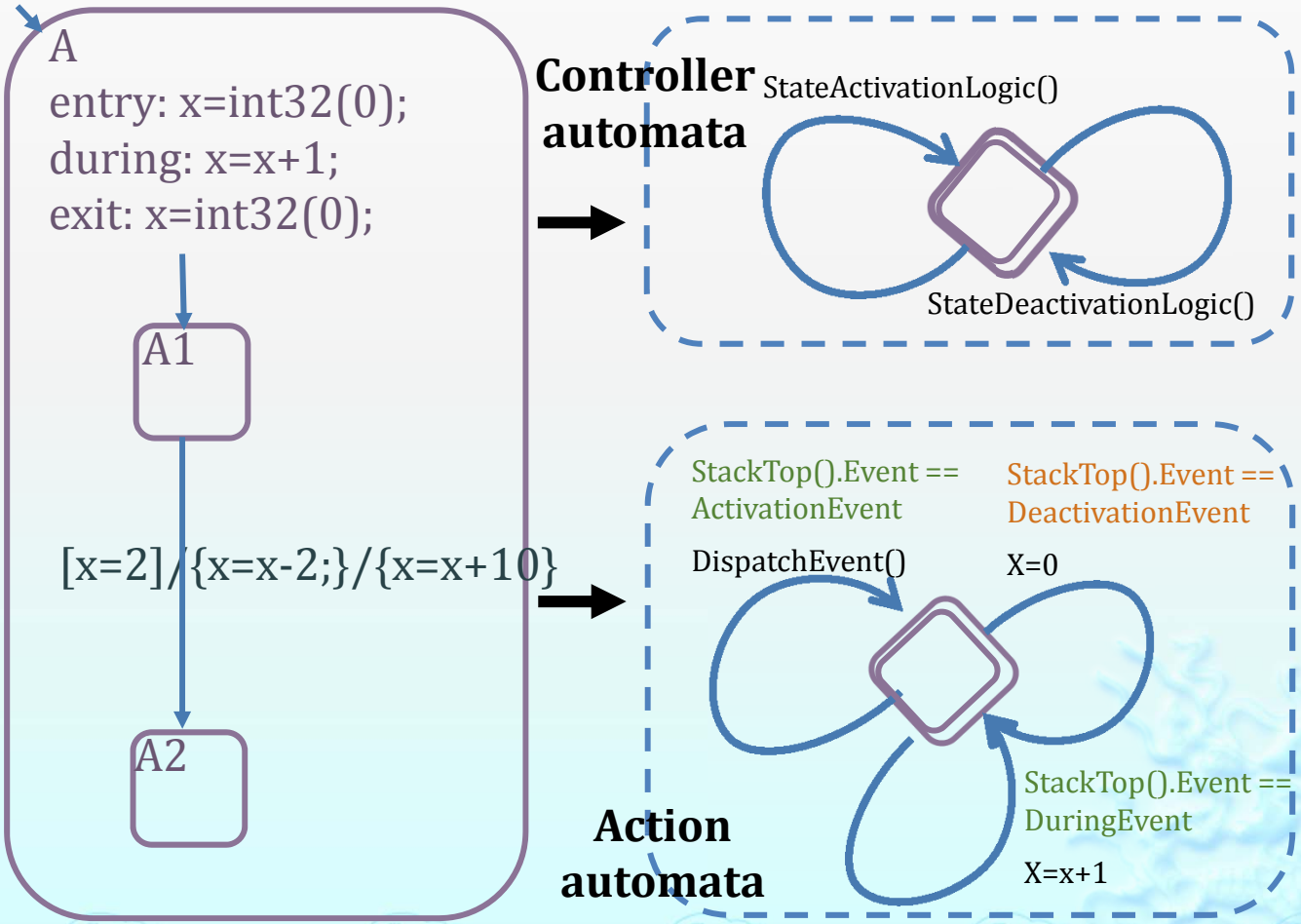
```
void DispatchDeactivationToChild(int state_s^f)
{
SubStates[ ] ← Substate(s^f);
if (SubStates[ ] are active) then
   if (SubStates[ ] are in parallel) then
      while (i ≤ length.PrioritySort(SubStates[ ])) do
         DispatchDeactivationToChild(SubStates[i]);
         i++;
      end while
   else
      DispatchEvent(Event DeactivationEvent);
   end if
end if
}
```

```
void HandleDeactivation(int state_s^f)
{
if (s^f has attached exit action) then
   Event DeactivationEvent.AutomatonType = action;
   DispatchEvent(Event DeactivationEvent).
end if
if (s^f is a sub-state) then
   int UpperLevelDest = AutomatonID(Parstate(s^f));
   Event DeactivationEvent.Dest = UpperLevelDest;
   DispatchEvent(Event DeactivationEvent);
end if
}
```

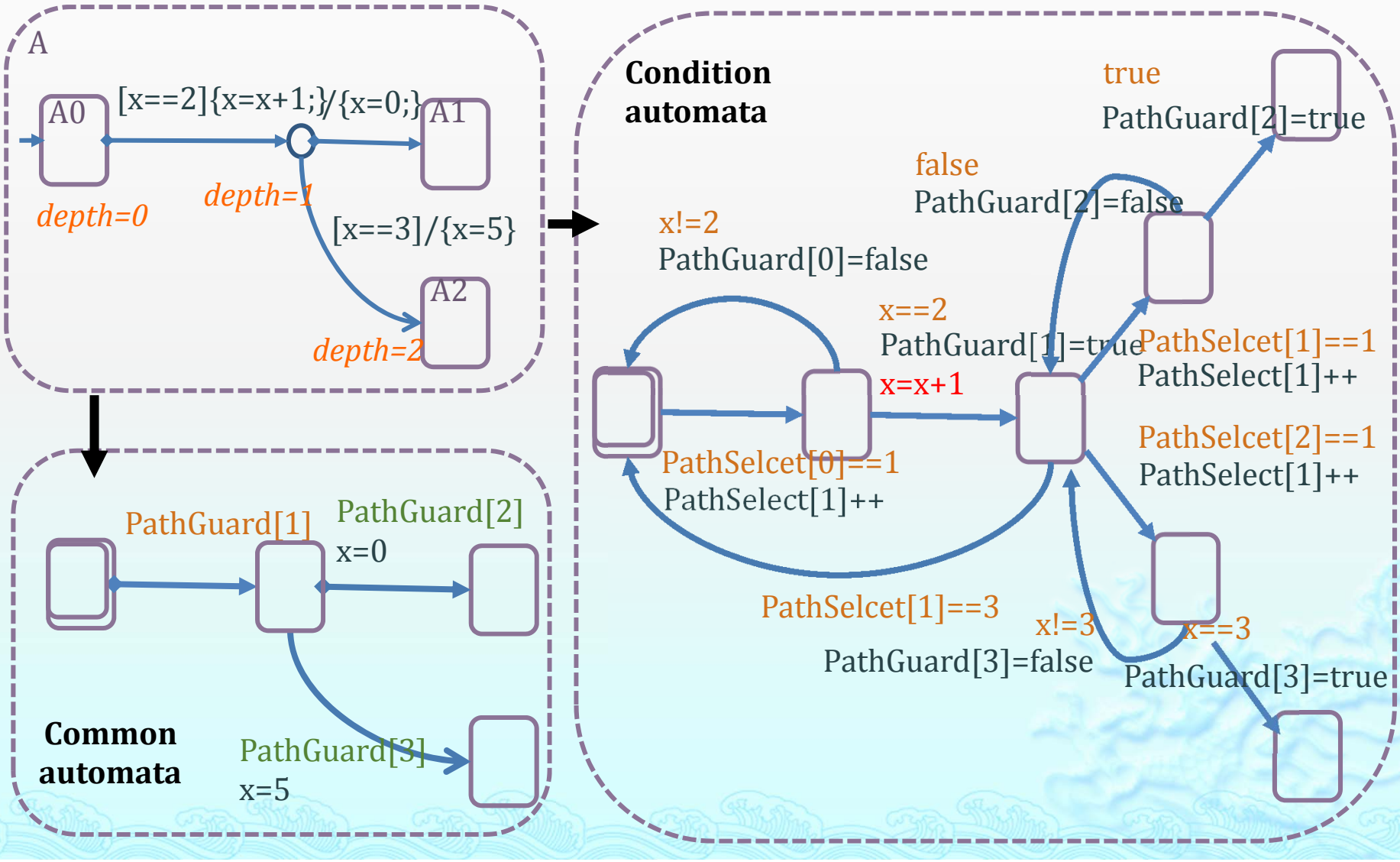
Stateflow to Timed automata-General principle

- For those composite states with decomposition or attached actions, we need to translate it to four cooperative parallel automata:
  - ***Controller automata:*** to simulate the event processing mechanism. Such as Deactivation( ).
  - ***Action automata:*** is responsible for handling the three kinds of attached actions (entry, during, exit).
  - ***Condition automata:*** is used to execute the conditional action, handle the junction, test the guard and priority contained in this composite state, and store the Boolean results.
  - ***Common automata:*** is used to execute the transitional action, and read the guard related array initialized by condition automata to execute the satisfied.

Stateflow to Timed automata- Composite state

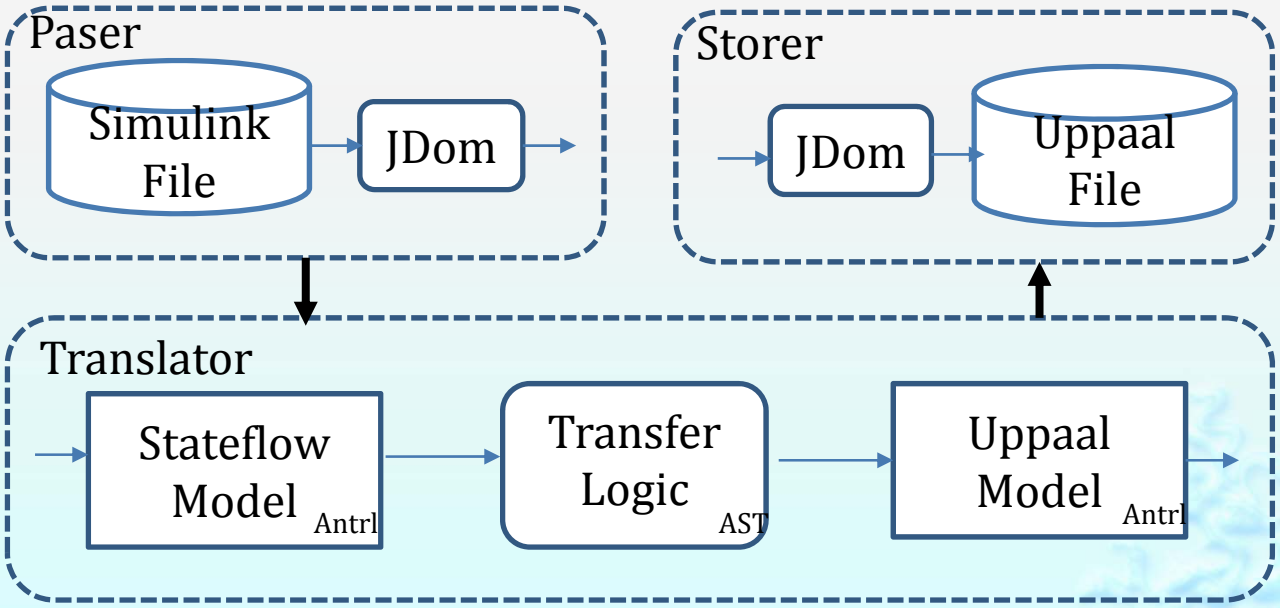


Stateflow to Timed automata-single state with transitions



Stateflow to Timed automata-Tool implementation

- Based on above transition rules, we implement a tool for automatically translation from Stateflow to Uppaal timed automata.



Tool Download:
https://www.dropbox.com/sh/374gcfjfei4ywlt/AACF9xqijvY-8nteIhcShIy9a?dl=0

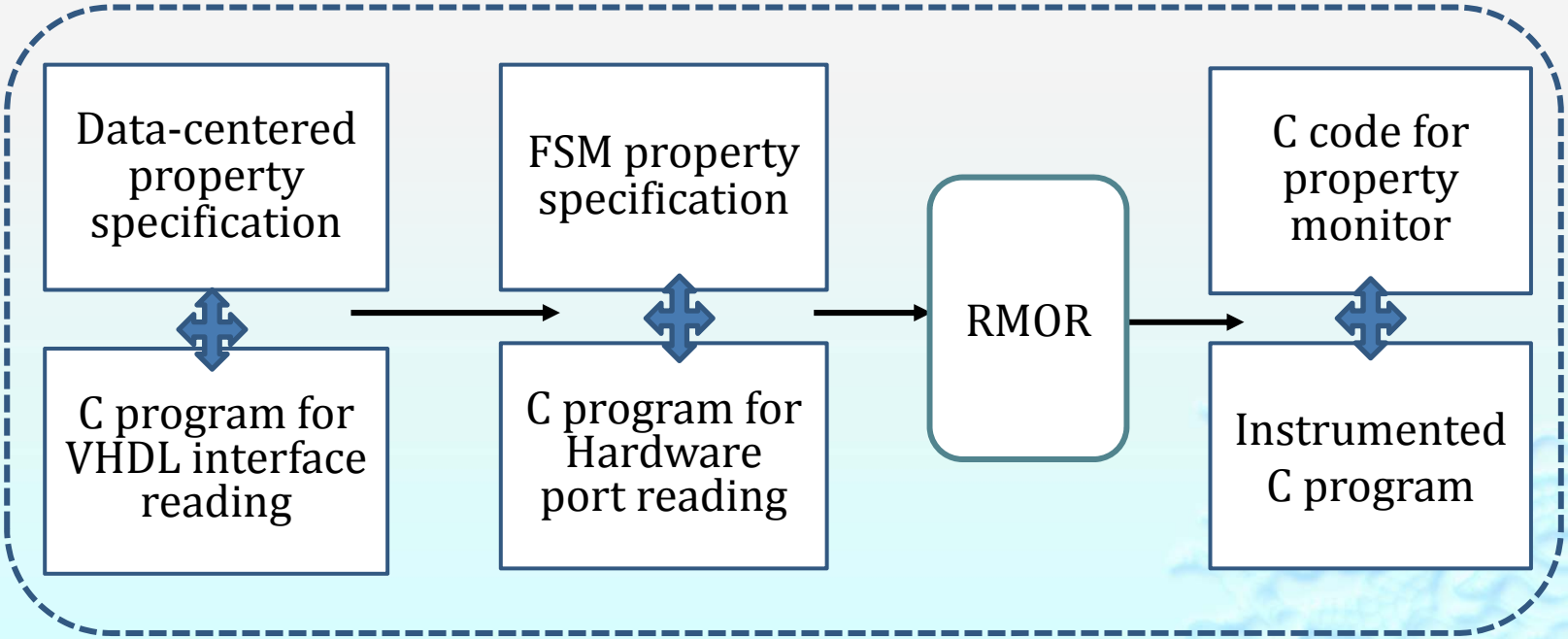
Runtime Verification Customization-Challenge

- The key technical ingredient in runtime verification for Stateflow is :
  - to specify dynamic runtime related properties that couldn't be easily described based on the abstract Stateflow model.
  - to choose proper run-time monitoring tools according to generated code of C and VHDL.

Reducing the complexity by customizing a data-centered runtime verification technique, into existing software monitor for runtime verification of VHDL .

Runtime Verification Customization-RMOR

- Based on the observation that VHDL has well defined interface of input and output data ports, we customize a data-centered runtime verification technique, into software monitor for runtime verification of VHDL



Approach-Enhanced Verification Conclusion

- Enhancement of Simulink Design Verifier

  Developed Stateflow to Uppaal tool for a more comprehensive verification of Stateflow model.

- Enhancement of Simulink Polyspace

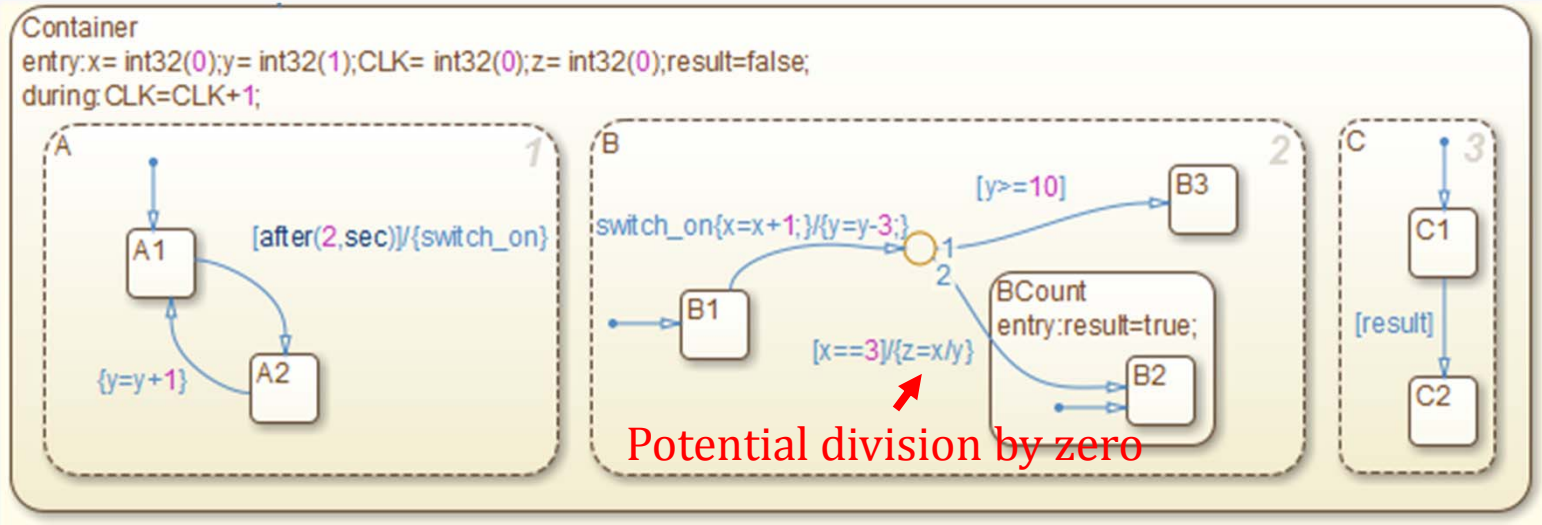
  Customized RMOR tool for a more comprehensive verification for generated C and VHDL code.

Outline

- **Motivation**
- **Background**
- **Framework**
- **Experiments**
  - Real train control system design
- **Conclusion**

Experiment-Division by Zero

- Recall the division by zero of Motivation



- Translate the model into Uppaal, and input the following property:

Property	Formula	Time(second)
P1	E<>Process_Chart_Container_B.SSID49 and Chart_y == 0 and Chart_x == 3	0.01

Experiment-Division by Zero

- The property consists of a serial combination of previous predicates, and means that y may be set to be 0 when the transition is enabled, which will cause the error of division by 0. Verification result shows that the property is satisfied and the error can be triggered.

Stateflow model, Translated Uppaal Model , and Property Specification Download:
https://www.dropbox.com/sh/374gcfjfei4ywlt/AACF9xqijvY-8nteIhcShIy9a?dl=0

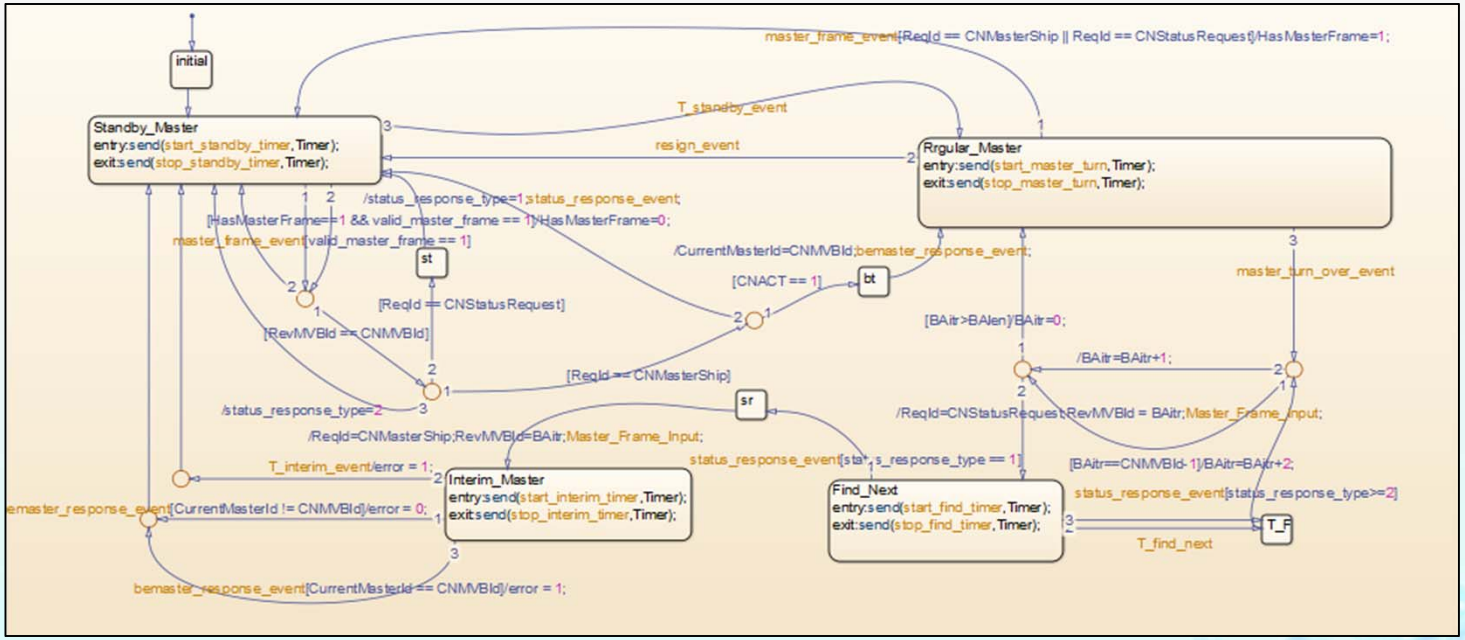
Experiment-Real Train Control System Design

- We apply the proposed approach to a real industrial application of Stateflow based model driven development of train communication control system.

- Given master rotation of vehicle bus controller as an example, the master transfer logic described in page 260 and Figure 105 of IEC 61375-1 are modeled as Stateflow model.

Experiment-Real Train Control System Design

- Stateflow model of the master rotation, in the bus master and slave rotation process, there may be inconsistence such that two (or no) masters appear at the same time.



Stateflow model, Translated Uppaal Model , and Property Specification Download: https://www.dropbox.com/sh/374gcfjfei4ywlt/AACF9xqijvY-8nteIhcShIy9a?dl=0

Experiment-Real Train Control System Design

- Translate the model, and input the following properties about no master collision:

Property	Formula	Time(second)
P1	A[ ] Process_Chart_OneMVB1(2)_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master and Process_Chart_OneMVB2(1)_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	32.93
P2	A[ ] not (Process_Chart_OneMVB1_LOGIC .Chart_OneMVB1_LOGIC_Rrgular_Master) and Process_Chart_OneMVB2_LOGIC .Chart_OneMVB2_LOGIC_Rrgular_Master	29.34
P3	A[ ] not (Process_Chart_OneMVB1_LOGIC .Chart_OneMVB1_LOGIC_Standby_Master) and Process_Chart_OneMVB2_LOGIC .Chart_OneMVB2_LOGIC_Standby_Master	33.02

Experiment-Real Train Control System Design

- Also described in IEC standard, the time constraint on an master controller between the finish of a master frame sending and the start of a response slave frame receiving should be less than 42.7us.

- The properties are not easy to capture in model level, because it is not easy to model dynamic transmission delay of data on bus in Stateflow, even with a preliminary channel model.

Experiment-Real Train Control System Design

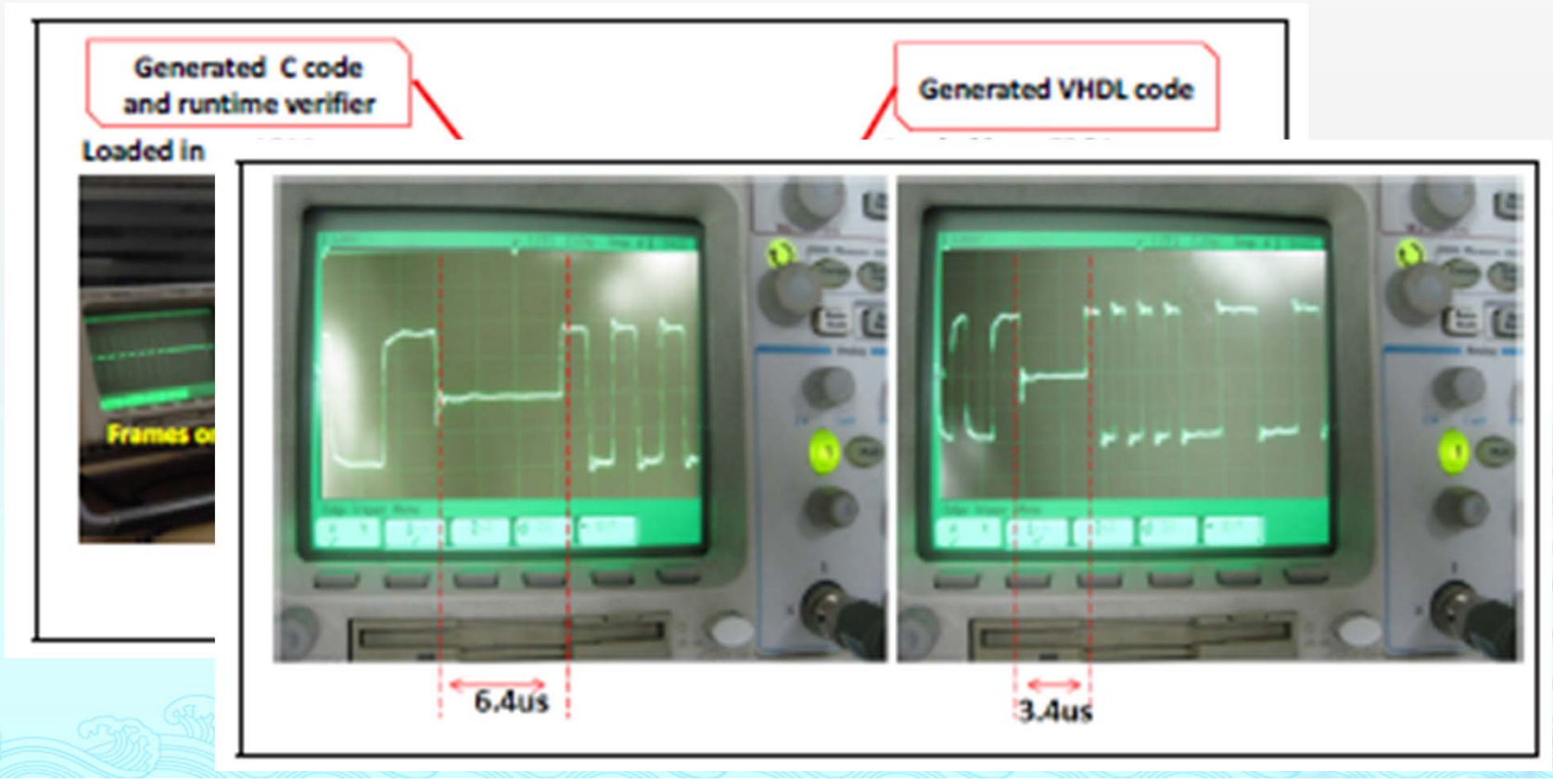
- After code generation, we can formalize the time constraint into a runtime verifier:

```
DataCenter Monitor TimeConstraints()
{
    ......
    event TimeoutReply =
     ((T_Master_Receive - T_Slav_Send_)<4)
    event TimeoutResponse =
     ((T_Master_Send - T_Slav_Receive)<42.7)
    event Trigger =
      TimeoutReply || TimeoutResponse;

    state safe{
     When Trigger -> error;
    }
}
```

Experiment-Real Train Control System Design

- We use oscilloscope to test the result that we get from the verifier.



Outline

- Motivation
- Background
- Framework
- Experiments
- Conclusion

Conclusion

- We present an approach to address the verification challenge of Stateflow based model driven development. By translating Stateflow model to timed automata, and customizing runtime monitors to generated codes of Stateflow model.

Conclusion

- Translated timed automata are about 6 times larger than the original Stateflow model, in terms of state and transition numbers. This is mainly caused by complex event stack of Stateflow, and hieratical, crossover and interruptible execution logic. *Using a simple semantics to express a rich semantics is not easy and increase complexity.*

- We plan to optimize our translating strategy to get more compassed timed automata and add some position information to make the translated timed automata well displayed in Uppaal

Thank you very much

Q/A?

For more details, please send email to
jiangyu198964@126.com