Formal V & V proposals for openETCS models

Xavier Crégut, Arnaud Dieumegard, Ning Gé, Marc Pantel and Andres Toom

University of Toulouse - IRIT/INPT - France
{Ning.Ge, Marc.Pantel, Xavier.Crégut}@enseeiht.fr

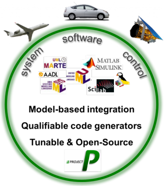
openETCS WP7 meeting
16/04/2013

Outline

1. Functional behavior with Simulink/Stateflow (or Scade) models
   - Context of the study
   - Verification by annotation generation
   - End to end verification
   - Conclusion and future work
2. Timed behavioral properties for SysML/UML-MARTE models
   - Case Study
   - SysML/UML-MARTE Time Properties Verification Framework
   - Conclusion

Plan

Practical Context

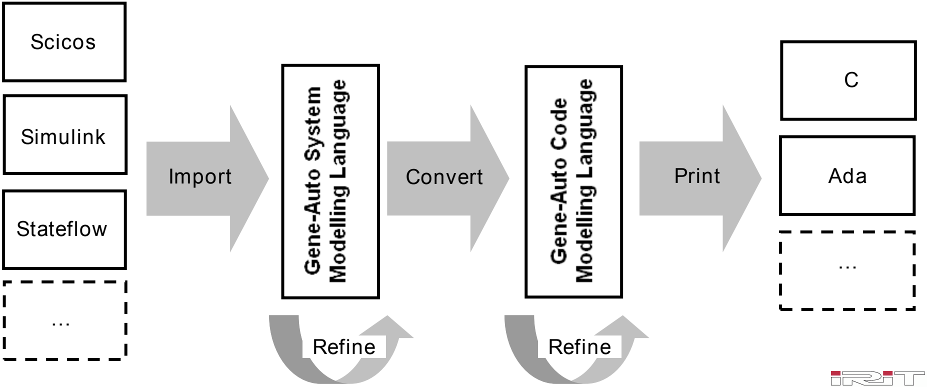


- Context
  - Automated code generation
  - Safety critical system certification
    - DO178/ED12, IEC61508, ISO26262, ECSS, . . .
- Purpose
  - Verification of the generated code functional correctness according to the generator specification
- Finality
  - Reduce anomaly risks and amount of tests for source code verification when using automated code generation

This work has been done in the context of the OPEES project

GENEAUTO (IB Krates, Alyotech, IRIT, AdaCore)

- Automated code generator from Simulink-Stateflow / Scicos to C / ADA / Java
- Verification process combining formal (Coq proof assistant [1]) and classic approaches (tests, proofreading)

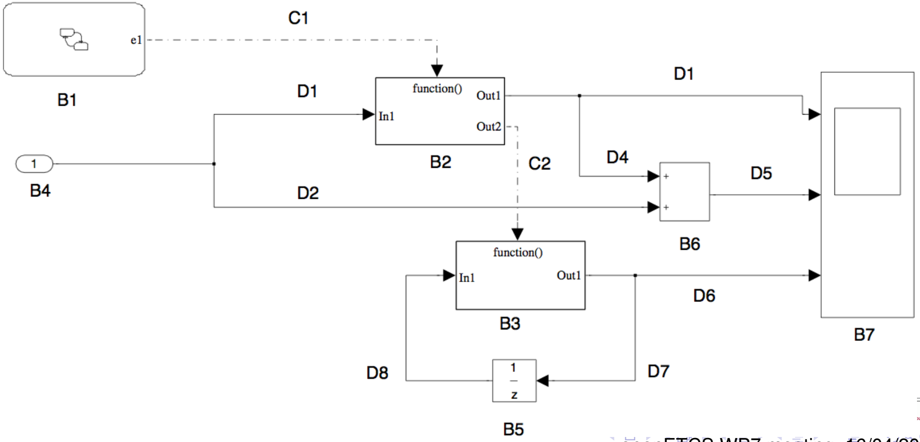


1. N.Izerrouken PhD thesis

Simulink/Scicos

- Modeling and simulation environment for systems

$\Rightarrow$ Command and control algorithms



Translation validation

- Introduced by A.Pnueli (1998)
- Verification of translators (compilers, code generators)
- Verification done at each run of the translator on the generated code
- Need :
  1. Formal verification elements for the source and target
  2. A way to compare these verification elements to assess the correctness

Annotations ?

- Descriptive elements (comments, . . .) on a code
  - No change of the code behavior
  - Not compiled or interpreted
- ACSL : ANSI/C Specification Language
  - Formally defined language used to describe code properties (behavior) for C programs
  - Implementation of the Hoare logic
    1. Pre/Post conditions
    2. Variants and Invariants (loops)
  - Weakest precondition calculus and SMT solvers ⇒ synthesis of the correctness proof
- Ada 2012/Spark Ada (see D. Mentre experiments)

Main questions

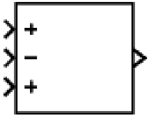
- How to verify an automated code generation ?
- How to handle the industrial constraints ?
- How to reduce the costs of the use of formal verification for common engineers ?

Principle of the approach

- Specification of the SIMULINK block library
- Extension of the GENEAUTO formal specification process
  - Define a formal block semantics through the properties it must satisfy (axiomatic semantics)
- Static analysis based verification (deductive kind)
  - Annotation generation
  - FRAMA-C (INRIA, CEA) : Weakest preconditions calculus combined with automated SMT solvers (Satisfaction Modulo Theories)

Specification

- Generic specification of an elementary SIMULINK block
  - $\mathbb{T}$ the type of input data
  - $I = \{X_i\}$ the set of inputs
  - $O = \{O_j\}$ the set of outputs
  - n (resp. p) $\in \mathbb{N}$ the number of input (resp. output) signals
  - $d_{in}, e_{in} / d_{out}, e_{out} \in \mathbb{N}$ the dimensions of the input/output signals
- The SIMULINK Sum block parameters
  - n inputs
  - a variant for each input (positive/negative)



Sum

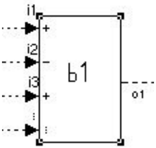
Specification

- Partial specification of the Sum block (for input and output as vectors)
  - Input :
    $n > 1, \forall i \in [1,n], \exists j \in \{1,d\}, \exists k \in [1,j], a_{i,k} \in \mathbb{T},$
    $$I = \left\{ X_i = \begin{pmatrix} a_{i,1} \\ \vdots \\ a_{i,j} \end{pmatrix} \right\}$$
  - Output :
    $$O = \left\{ \forall i \in [1,n], \triangle_i \in \{+,-\}, \begin{pmatrix} \sum_{i=1}^{n} \triangle_i a_{i,1} \\ \vdots \\ \sum_{i=1}^{n} \triangle_i a_{i,j} \end{pmatrix} \right\}$$

Annotations generation : pre-conditions and post-conditions

- Specific additional annotations for instructions (conditionals, loops, ...)
- Annotated generated code :

```
/*@ requires \forall integer m;
      0 <= m < vector_size ==>
      \valid(&b1.o1+m) &&
      \valid(&b1.i1+m) && ... ;
   ensures \forall integer m; 0 <= m < vector_size ==>
      b1.o1[m] == b1.i1[m] - b1.i2[m] + ... ;
*/
/*@ loop invariant 0 <= index <= vector_size;
   loop invariant \forall integer m;
      index <= m < vector_size ==>
        b1.o1[m] == \at(b1.o1[m],Pre);
   loop invariant \forall integer m; 0 <= m < index ==>
      b1.o1[m] == b1.i1[m] - b1.i2[m] + ...;
   loop variant vector_size - index;
*/
for (int index = 0; index < vector_size; index++){
    b1.o1[index] = b1.i1[index] - b1.i2[index] + ...;
}
```



Pre/Post conditions and loop variants/invariants

- Pre-conditions

```
requires \forall integer m;
  0 <= m < vector_size ==>
    \valid(&b1.o1+m) &&
    \valid(&b1.i1+m) && ... ;
```

- Post-condition

```
ensures \forall integer m; 0 <= m < vector_size ==>
  b1.o1[m] == b1.i1[m] − b1.i2[m] + ... ;
```

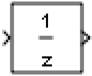
- Additional annotations

```
loop invariant 0 <= index <= vector_size;
loop invariant \forall integer m; index <= m < vector_size ==>
  b1.o1[m] == \at(b1.o1[m],Pre);
loop invariant \forall integer m; 0 <= m < index ==>
  b1.o1[m] == b1.i1[m] − b1.i2[m] + ...;
loop variant vector_size − index;
```

Memory blocks 1/2

- Initialization of the block (computed at system init)

```
/*@ requires \valid(&mem.d_mem);
    ensures \forall integer n; 0 <= n < 2 ==>
   mem.d_mem[n] == 0.0;
*/
void init(){
  /*@ loop invariant 0 <= i <= 2;
      loop invariant init_mem_invariant:
        \forall integer n; 0 <= n < i ==> mem.d_mem[n] == 0.0;
      loop variant 2-i;
  */
  for (int i=0;i<2;i++){
    mem.d_mem[i] = 0.0;
  }
}
```

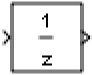


Memory blocks 2/2

- Computation of the block

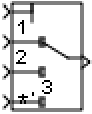
```
//@ ghost double pre_mem[2];

/*@ requires \forall integer n; 0 <= n < 2 ==>
      \valid(&b.d_in+n) && \valid(&b.d_out+n)
      && \valid(&mem.d_mem+n);
    ensures \forall integer n; 0 <= n < 2 ==>
      b.d_out[n] == pre_mem[n] &&
      mem.d_mem[n] == b.d_in[n];
 */
void compute(){
  /*@ loop invariant 0 <= i <= 2;
      loop invariant get_mem_invariant:
        \forall integer n; 0 <= n < i ==>
          b.d_out[n] == pre_mem[n];
      loop invariant set_mem_invariant:
        \forall integer n; 0 <= n < i ==>
          mem.d_mem[n] == b.d_in[n];
      loop variant 2-i;
  */
  for (int i=0;i<2;i++){
    b.d_out[i] = mem.d_mem[i];
    //@ ghost pre_mem[i] = mem.d_mem[i];
    mem.d_mem[i] = b.d_in[i];
  }
}
```



Control structure blocks

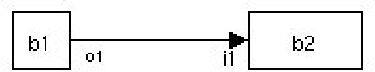
```
/*@ ensures \forall int m; ((int)i1) == 2 ==>
      (0 <= m < 2 ==> o1_vect[m] == i2_vect[m]);
    ensures \forall int m; ((int)i1) == 3 ==>
      (0 <= m < 2 ==> o1_vect[m] == i3_vect[m]);
*/
switch((int)i1) {
  case 2 : {
    /*@ loop invariant ((int)i1) == 2;
        loop invariant 0 <= i <= 2;
        loop invariant \forall int m; 0 <= m < i ==>
          o1_vect[m] == i2_vect[m];
        loop variant 2-i;
    */
    for (int i = 0; i < 2; i++) {
      o1_vect[i] = i2_vect[i];
    }
    break;}
  case 3 : {
    /*@ loop invariant ((int)i1) == 3;
        loop invariant 0 <= i <= 2;
        loop invariant \forall int m; 0 <= m < i ==>
          o1_vect[m] == i3_vect[m];
        loop variant 2-i;
    */
    for (int i = 0; i < 2; i++) {
      o1_vect[i] = i3_vect[i];
    }
    break;}
}
```



Data flow annotations

- Annotations for the data flows between blocks

```
/*@  ensures \forall integer n; 0 <= n < vector_size ==>
       b2.i1[n] == b1.o1[n];
*/
{
  /*@  loop invariant 0 <= index <= vector_size;
       loop invariant \forall integer n; 0 <= n < index ==>
         b2.i1[n] == b1.o1[n];
       loop variant 2-index;
  */
  for(index=0; index < vector_size; ++index){
    b2.i1[index] = b1.o1[index];
  }
}
```



Work done on specific blocks

- Annotation writing for representative blocks
  - Computation blocks (Sum, Gain, Interpolators)
  - Memory block (Delay)
  - Control structure blocks (MultiPortSwitch, ForIterator, DelayRE)
- Verification of the annotated code with FRAMA-C

Verification

- Using FRAMA-C
  - WP/Jessie : proof obligations generation
  - SMT solvers : assess the satisfaction of the proof obligations

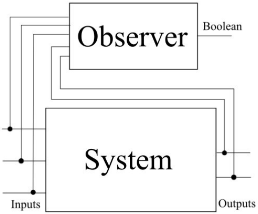
```
[  Valid  ] Function 'compute' ensures ∀ ℤ m; (0 ≤ m) ∧ (m < 4) ⇒
                               (S[m] ≡ x1[m]+x2[m])
[  Valid  ] Function 'compute' decrease: <TODO>
[  Valid  ] Function 'compute' loop invariant (0 ≤ index) ∧
                                 (index ≤ 4);
[  Valid  ] Function 'compute' loop invariant ∀ ℤ m;
                                 (index ≤ m) ∧ (m < 4) ⇒
                                 (S[m] ≡ \at(S[m],Pre));
[  Valid  ] Function 'compute' loop invariant ∀ ℤ m;
                                 (0 ≤ m) ∧ (m < index) ⇒
                                 (S[m] ≡ x1[m]+x2[m]);

-----------------------------------------------------------
  No proofs        :    0
  Partial proofs   :    0
  Complete proofs  :    6
  Total            :    6
```

$\Rightarrow$ Automated verification of the generated code according to the expected behavior (pre/post conditions for the blocks)

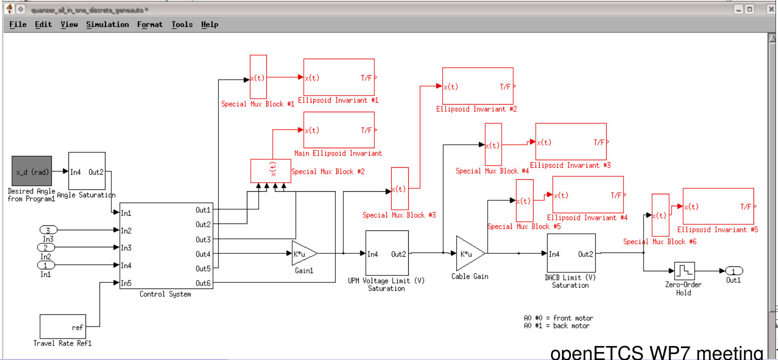
Using observers (experimented in GeneAuto)

- Expressing system properties (behavior/expected results) using system conception language (related to execution context, models of the external world, simulations)
- Observers : Pre/Post conditions of the whole system
- Translation of the observer as annotations (eg. Ellipsoid for Lyapunov stability)



Joint work with T. Wang, R. Jobredeau, P-L. Garoche, E. Feron (ONERA, Georgia Tech)

Annotated Quanser Model



Conclusion

- Verification of the generated code semantics correctness (conformance to the specification)
  - Pre/Post conditions for each block
  - Invariants for blocks and data flows
- Possibility to generate higher level properties (observers)
- Automation of the verification

$\Rightarrow$ Fully transparent proof for the user

Future work

- Perspectives
  - Definition of a user dedicated block library specification language [2] for automatic code/doc. generation
  - Transformation from the block library to the annotations backends
  - Invariants/Variants generation
    - Complexity / Efficiency
    - Automatically synthesized / Provided in the block specification
  - Integration in a qualified development processes
- Open questions :
  - Complete automation of the process
  - Scalability of the approach

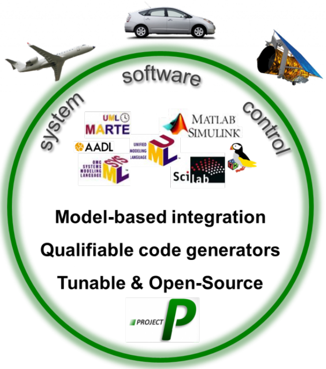
2. Joint work with A.Toom (IB Krates)

Thanks for your attention

Any questions ?

Plan

Research Context



- Use Model-Driven Engineering for SysML/UML-MARTE specifications
- Allow an early integration of feasibility analysis in the design phase
- Enable a rapid iterative design-prototype cycle
- Focus on the timing properties for the real-time embedded systems

Research Context

Objective

Rapidly verify that system's timing properties matches the specifications by model checking

Problematic

- How to transform the semi-formal language SysML/UML to formal verification language
- How to translate user required properties in a formal way
- How to verity property by model checking
- How to control the scalability of the approach to use in industry applications (Model checking core issue : combinatorial explosion of state space)

End User Requirements

Timing Properties

- Upper Loop Bound
- Best/Worst-Case Execution Time (Max/Min Absolute Time)
- Best/Worst-Case Traversal Time (Max/Min Time Interval)
- Best/Worst-Case Response Time (Max/Min Time Interval)
- Task level & event level time constraints (logical and physical)
  - Synchronization, Coincidence, Exclusion, Precedence, Sub-Occurrence, Causality

Time Petri Net

Time Petri Net

A Time Petri Net $\mathcal{T}$ is a tuple *(P, T, E, W, M_0, (α, β))*, supported by TINA [a].

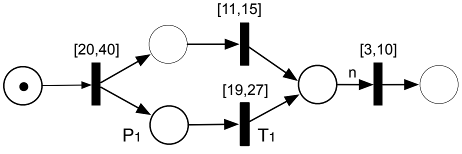
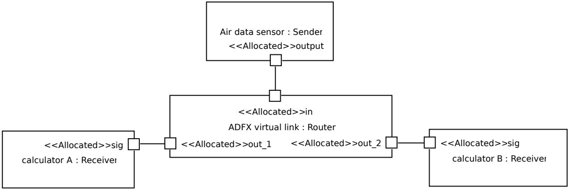


FIGURE : Time Petri Net Example

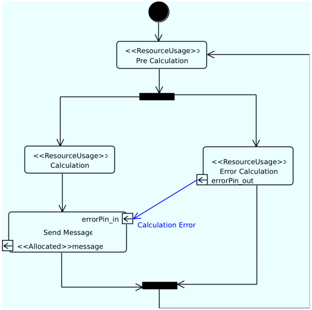
- TTS (Timed Transition Systems) is TPN with data handling
- Pre() : data constraints on the transition
- Act() : data manipulation when transition is fired

a. http ://projects.laas.fr/tina//

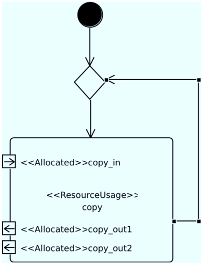
Case Study



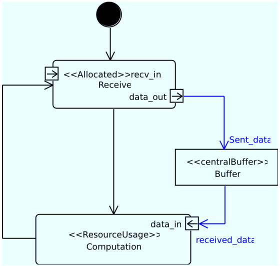
Sender



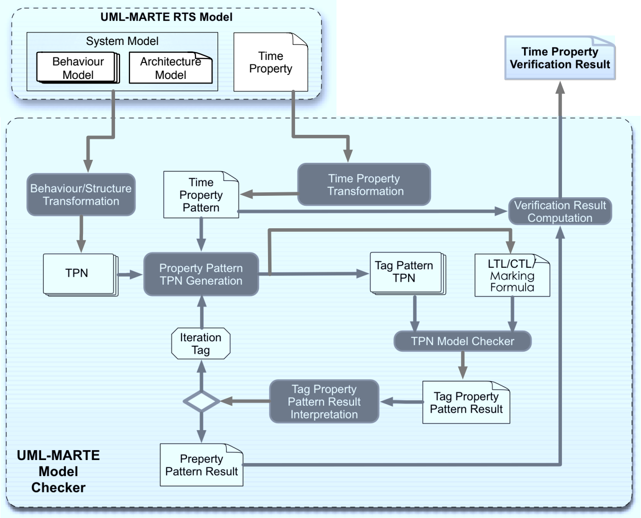
Router



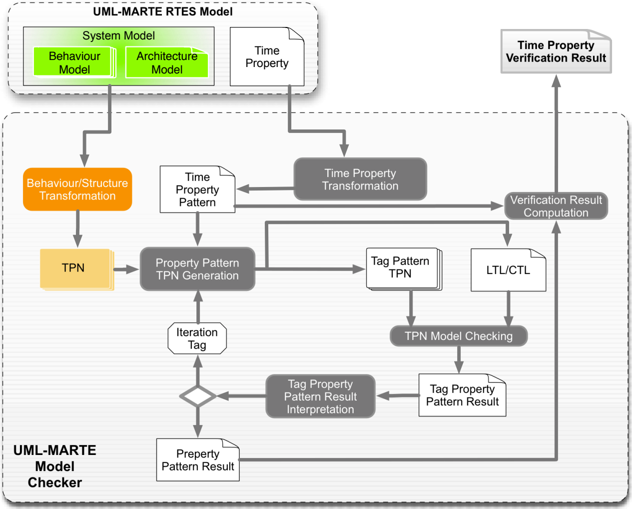
Receiver



Framework

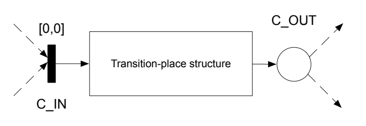

UML-MARTE RTS Model
System Model
Behaviour Model
Architecture Model
Time Property
Time Property Verification Result
Behaviour/Structure Transformation
Time Property Pattern
Time Property Transformation
Verification Result Computation
TPN
Property Pattern TPN Generation
Tag Pattern TPN
LTL/CTL/ Marking Formula
Iteration Tag
TPN Model Checker
Tag Property Pattern Result Interpretation
Tag Property Pattern Result
Property Pattern Result
UML-MARTE Model Checker

Transformation from SysML/UML-MARTE to Time Petri Nets

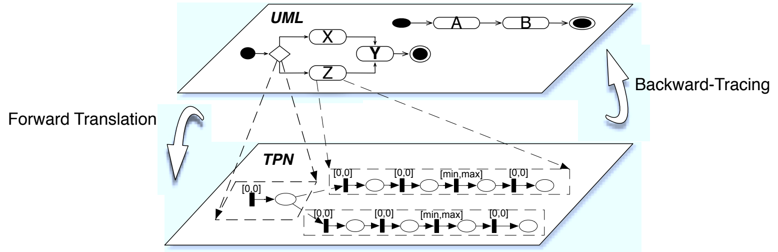


Transformation from SysML/UML-MARTE to Time Petri Nets

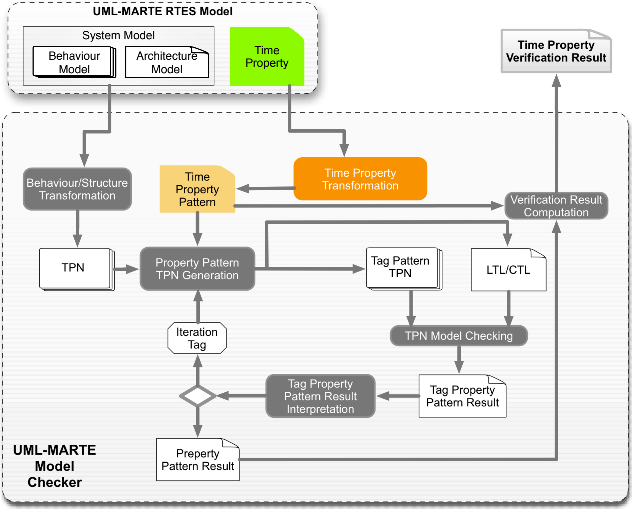
Translation SysML/UML into TPN : General Pattern



Translation SysML/UML into TPN : Traceability



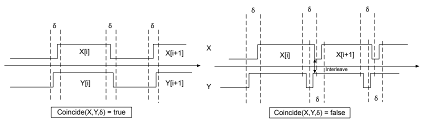
Translation Time Property into Time Property Patterns



Translation Time Property into Time Property Patterns

Coincidence(TaskA, TaskB, δ)

- Task X and Y are coincident iff. the n^{th} occurrence of X occurs simultaneously with the n^{th} occurrence of Y while $n \in \mathbb{N}$, within time tolerance δ. It is equivalent saying the n^{th} occurrence of X_s occurs simultaneously with the n^{th} occurrence of Y_s, and the n^{th} occurrence of X_e occurs simultaneously with the n^{th} occurrence of Y_e.



- $Coincide(X,Y,\delta) \equiv$

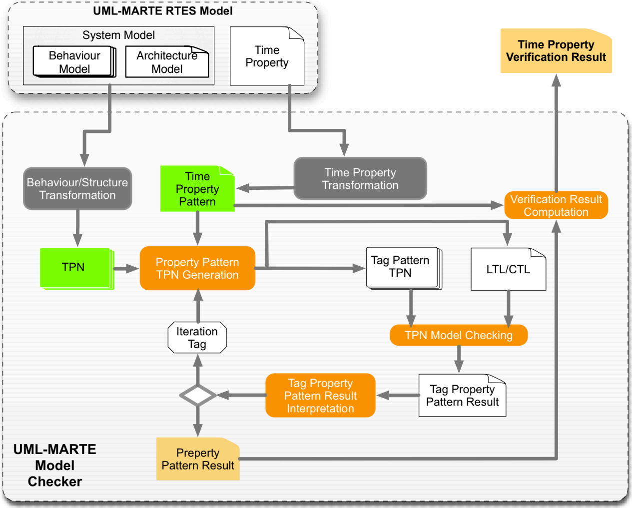
$$\forall t \in \mathbb{R}_+ : (|O(X_s^t) - O(Y_s^t)| < 2) \wedge (|O(X_e^t) - O(Y_e^t)| < 2) \quad (1)$$

$$\forall t \in \mathbb{R}_+ : (|T(X_s^t) - T(Y_s^t)| < \delta) \wedge (|T(X_e^t) - T(Y_e^t)| < \delta) \quad (2)$$

$$\forall i \in \mathbb{N}^* : (T(X_e^i) + \delta < T(Y_s^{i+1})) \wedge (T(Y_e^i) + \delta < T(X_s^{i+1})) \quad (3)$$

Formal Specification	Time Property Pattern
X_s^{i+1}	Representation of the next occurrence of event X_s^i
$\lvert O(X_a^t) - O(Y_a^t)\rvert < \delta$	Occurrence number difference between events X_a^t and Y_a^t
$\lvert T(X_a^t) - T(Y_a^t)\rvert < \delta$	Max time interval between events X_a^t and Y_a^t
$T(X_e^i) + \delta < T(Y_s^{i+1})$	Min time interval between events X_a^t and Y_a^t

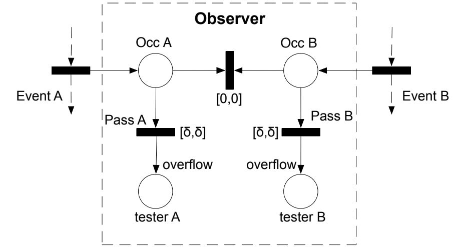
Verification of Time Property Patterns



Verification of Time Property Patterns

Verification of *Coincidence(TaskA, TaskB, δ)*

- Pattern $|T(a^t) - T(b^t)| < \delta$



- Existence of marking in reachability graph
- $\Diamond(\mathit{testerA} = 1) \vee \Diamond(\mathit{testerB} = 1)$

Feedback of Error-Source Location

Help the user analyse the failure of model-checking

- Violation-existing paradigm (i.g. deadlock)
- Desired-missing paradigm (i.g. dead branch)
- Analyse the dependency in reachability graph to locate the probable error-source locations using Hidden Markov Models
- Calculate the probability of each probable error-source location
- Return the error-sources in the design UML model with probability

Conclusion

Proposed Methods

- SysML/UML-MARTE Properties verification framework
  - *Time Properties Verification Framework for SysML/UML-MARTE Safety Critical Real-Time Systems (In proceedings of ECMFA'2012)*
- Transformation from SysML/UML-MARTE to Time Transition System (TPN & Data)
  - *Time Properties Dedicated Transformation from SysML/UML-MARTE Activity to Time Transition System (In proceedings of UML&FM'2012)*
- Formal specification of task-level time constraints by time property patterns Observer based model checking by TPN to assess time property patterns
  - *Formal Specification and Verification of Task Time Constraints for Real-Time Systems (In proceedings of ISoLA'2012)*

Future works

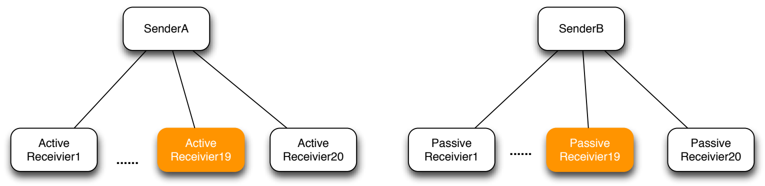
- Apply our method on more complex examples
- Extend the framework to cover more kinds of properties

Thanks. Question ?

Performance Evaluation of Case Study

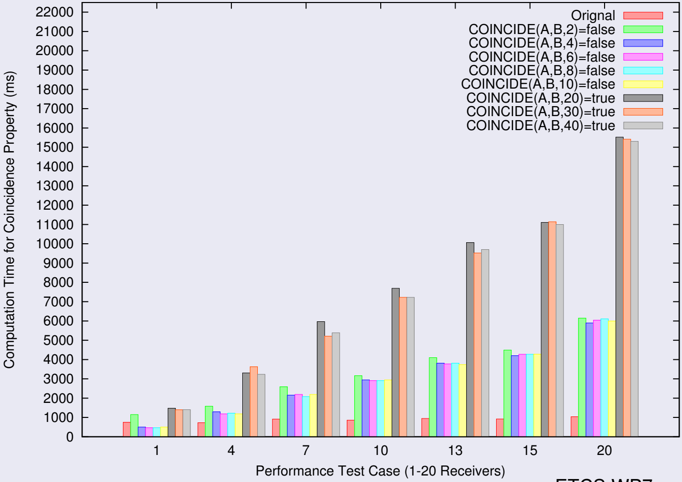
Scalability Test

A system with a scale of 2 senders, 1-20 pairs of active-passive receivers.



Performance Evaluation of Case Study

Result Analysis



Scalability

Problematic

- State space explosion problem of model checking

Guaranty Method

- Transformation method : Property-driven transformation
- Formal semantic of timing : Decomposition method and binary search method
- Optimization method : Optimizing TPN before verification
- Verification : Marking abstraction-level, on-the-fly (SIFT)

Transformation from SysML/UML-MARTE to TPN

Transformation Principle

- The transformation of one SysML/UML element may be different according to the time property to be assessed.
- For some intuitive elements not influencing time properties, the translated TPN semantic can be standardized and homogeneous for all the property.
- The transformation should guarantee the consistency of the semantics between high-level model and lower-lever model.
- The generated TPN models should be able to perform a highly efficient verification of time properties in large scale asynchronous applications.
- The transformed elements should facilitate the assembly, which may cause the performance a little degraded than manual modelled one.