On the Correctness of Model Transformations in the Development of Embedded Systems

Gabor Karsai, Anantha Narayanan, Sandeep Neema

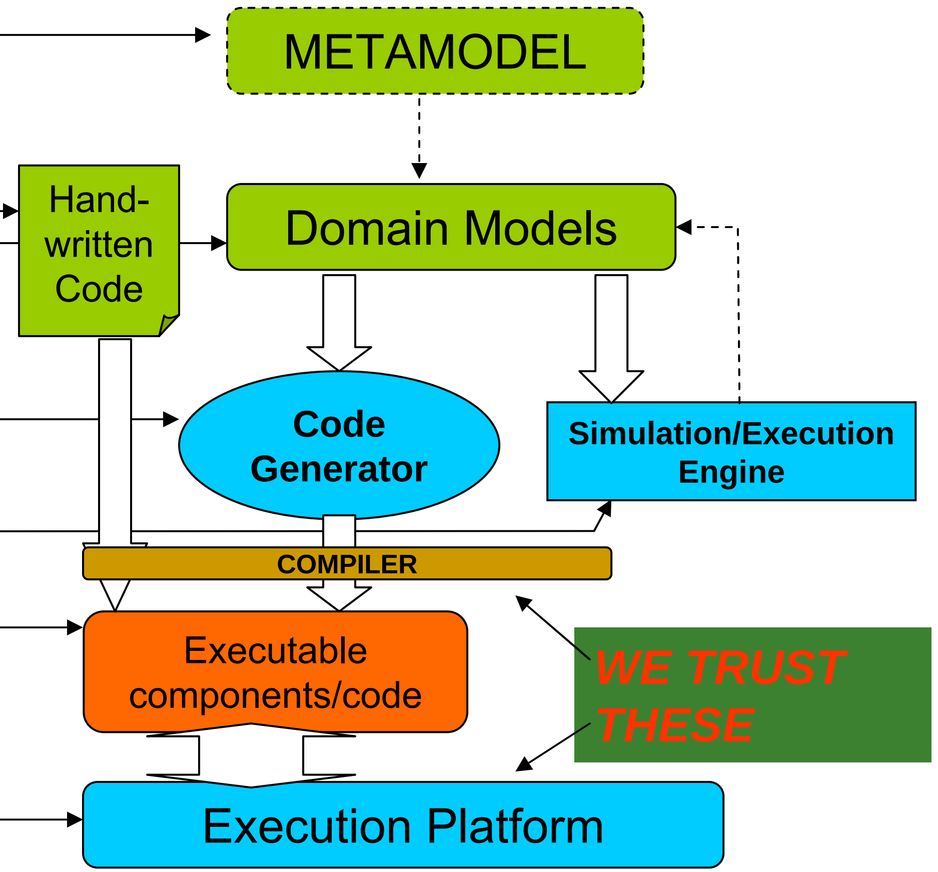
Institute for Software-Integrated Systems
Vanderbilt University
Nashville, TN 37235, USA

Overview

- The Problem
- Background: Instance-based verification
- Approaches:
  - Certification through bisimilarity checking
  - Certification via semantic anchoring
- Exercise problem:
  - Show the non-existence of infinite recursion
- Summary

Model-based Embedded Software Development Today

- Defines the modeling language (document)
- The “source code”
- The “compiler”
- The “verification tool”
- The “code”
- The “OS”

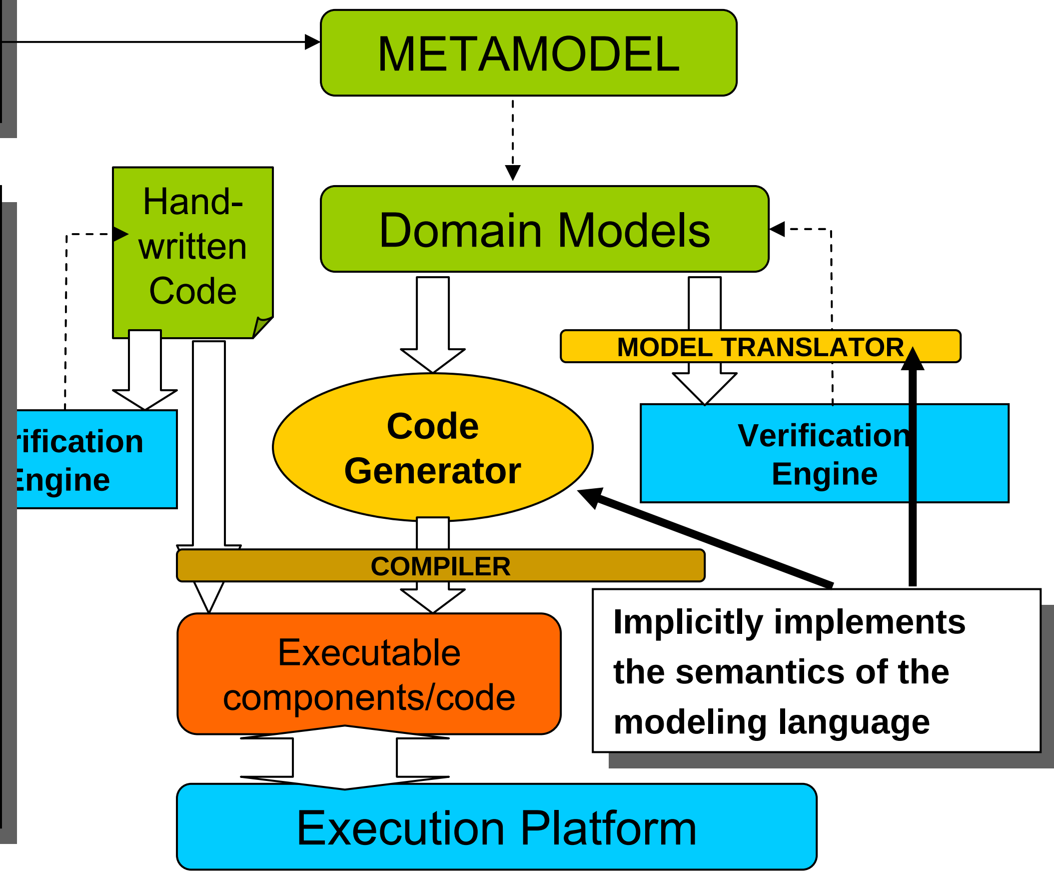


Model-based Software Development Near Future

- Formally defines the modeling language

Essential questions for model-based development:

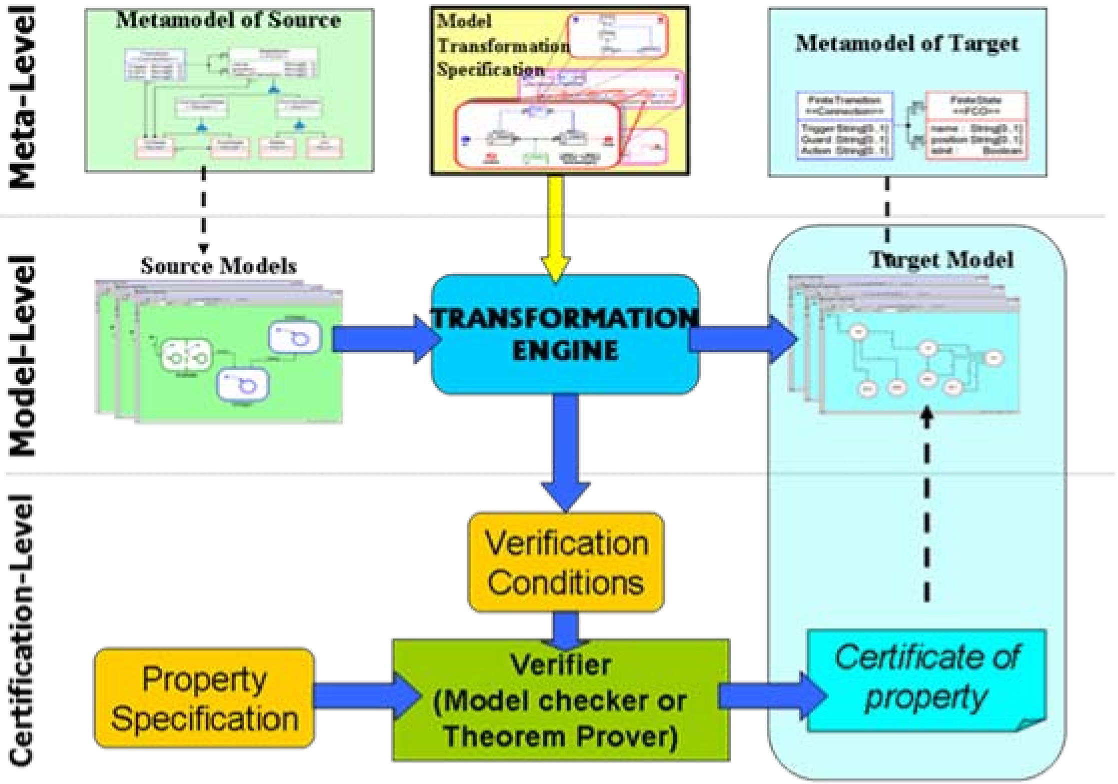
1. How do you know that your model transformations (model translator/code generator) are correct?
2. How do you know that the products of the verification engine are true for the generated code running on the platform?



Background: Instance-based Verification

Instance-based generation of certificates: (NASA/ARC/RSE)

1. Use the transformation engine to co-generate 'verification conditions'
2. Use a theorem prover/model checker to check properties on the verification conditions

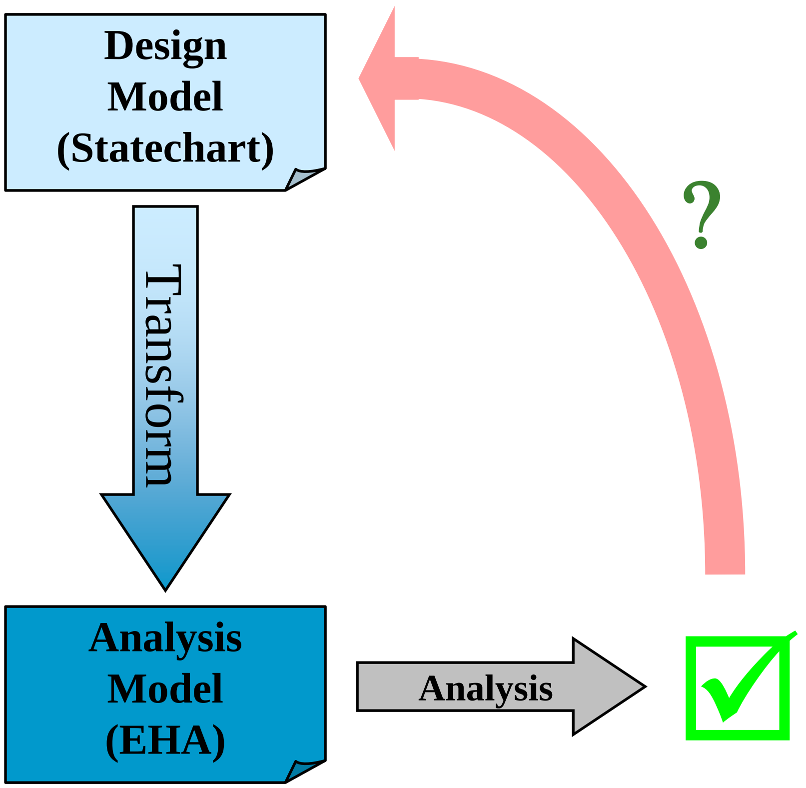


Approaches (1): Certification through bisimilarity checking

- Problem description:
  - Statechart to EHA transformation
- Bisimulation
- Checking bisimulation between Statechart and EHA models

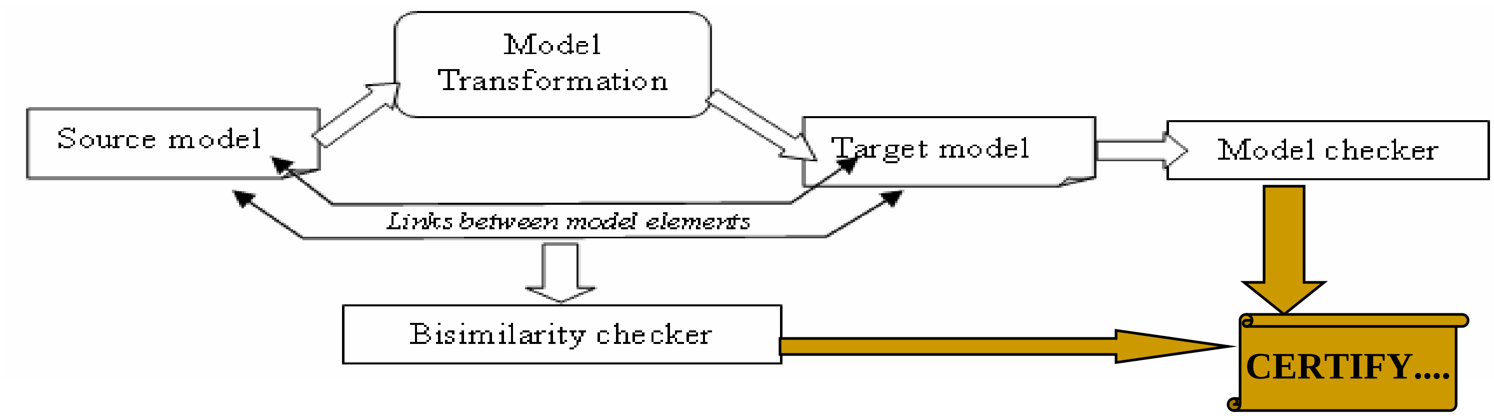
Problem Description: Analysis of Design Models

- Correctness of Model Transformations is central to the success of a model driven development process
- Systems are designed using a design language, and transformed into an analysis language for analysis
- The results of the analysis hold on the analysis model
- They will hold on the design model only if the transformation *preserved* the semantics with respect to the property of interest



Verifying Transformations

- Checking whether a transformation preserves
  - Certain properties of interest
  - For a certain instance
  - Using bisimulation



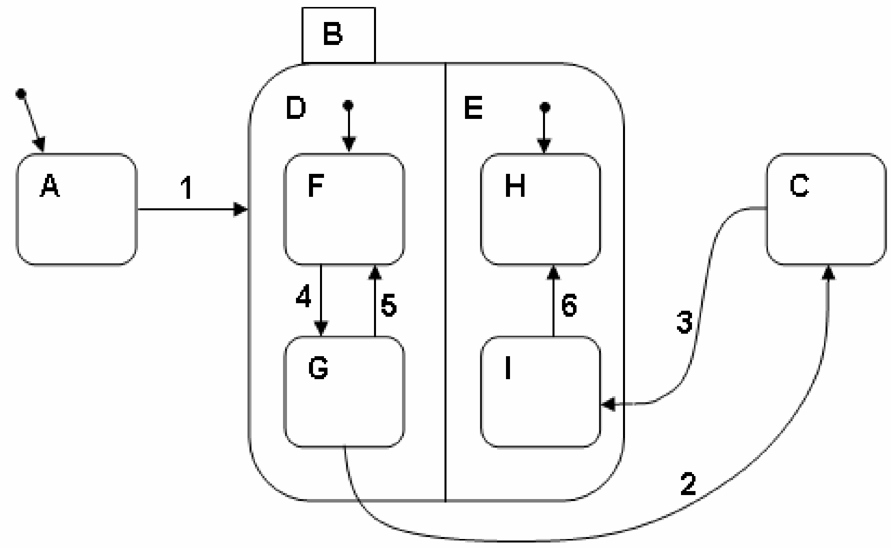
- We can certify that the analysis results are valid on the design model for this instance
- We do not attempt to prove the general correctness of the transformation itself

Bisimulation

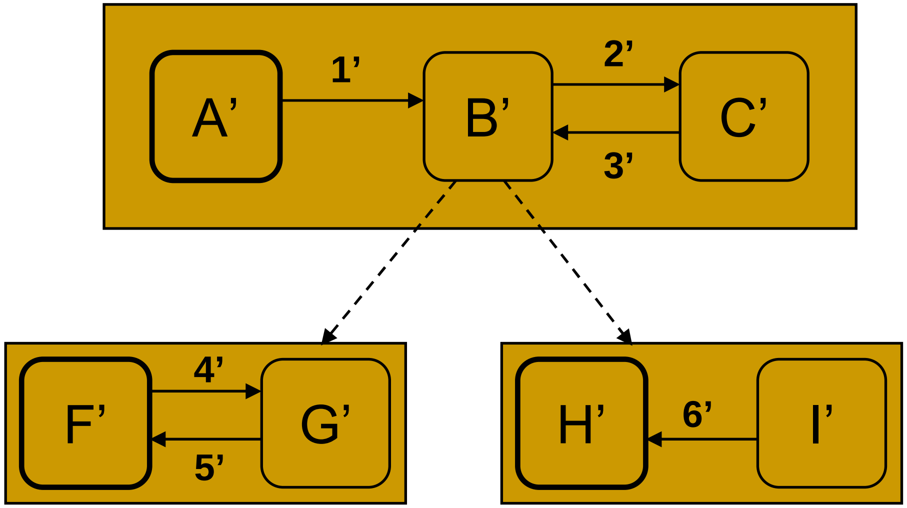
- Given a labeled state transition system (**S**, Λ, →), a bisimulation relation is a binary relation **R** such that
  - For every pair of elements *p*, *q* in **S**, if (*p*, *q*) is in **R**
  - For all α in Λ, and for all *p'* in **S**
  - $p \xrightarrow{\alpha} p'$ implies that there is a *q'* in **S** such that
  - $q \xrightarrow{\alpha} q'$ and (*p'*, *q'*) is in **R**
  - And for all *q'* in **S**
  - $q \xrightarrow{\alpha} q'$ implies that there is a *p'* in **S** such that
  - $p \xrightarrow{\alpha} p'$ and (*p'*, *q'*) is in **R**
- Use **cross-links** to trace the relation **R**, and check if it is a bisimulation

Statechart to EHA Transformation

Source - Statechart



Target - EHA

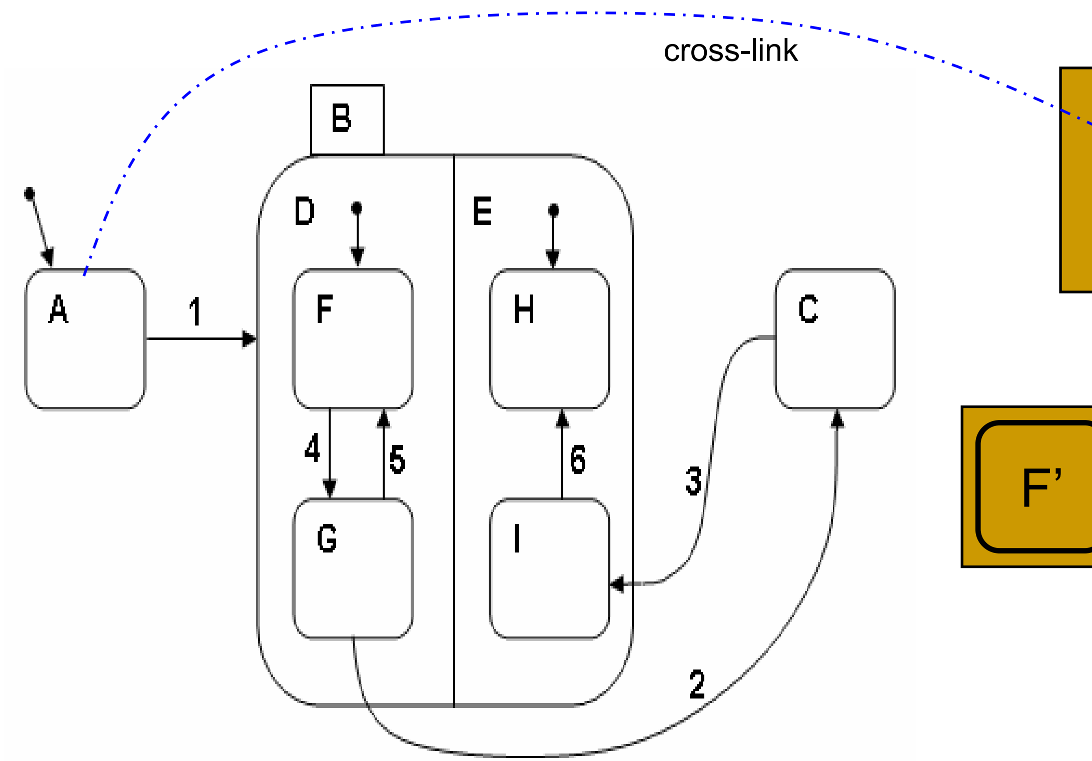


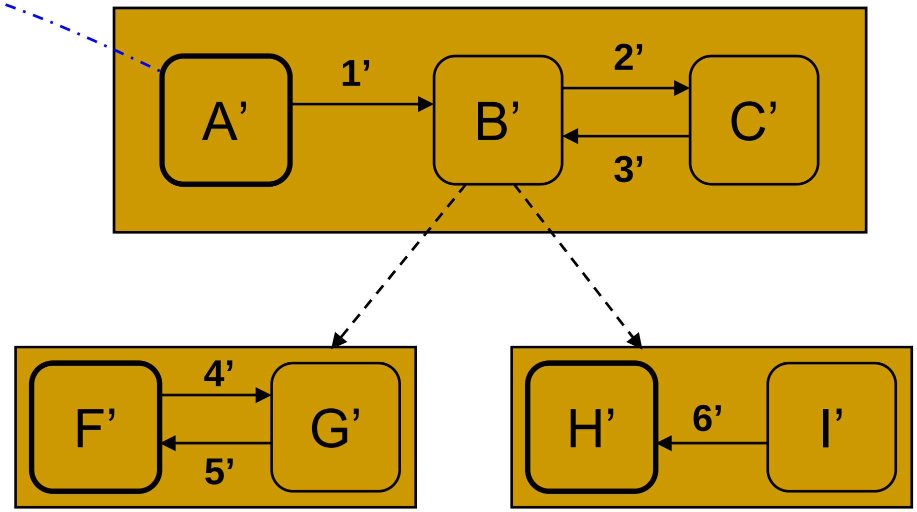
Transition Label	SR	TD
2'	G'	
3'		I', F'

Statechart to EHA Transformation

Source - Statechart

Target - EHA





Transition Label	SR	TD
2'	G'	
3'		I', F'

[illegible] Sequential Automata

Verifying the Transformation

- When the target elements are created, we know what source elements they correspond to
- But we do not know whether
  - all the source elements were considered
  - all compound states were refined correctly
  - all transitions were connected between the correct corresponding elements
  - all inter-level transitions were annotated correctly
- To verify these conditions, we check if the two models are bisimilar
  - Using the cross-links to trace the equivalence relation R

Statecharts and EHA

- State Configuration – A maximal set of states that a system can be active in simultaneously
  - Closed upwards
- Transitions – Take the system from one state configuration to another
- Two state configurations S_1 and S_2 are in **R** if
  - every state s_1 in S_1 has a state s_2 in S_2 and (s_1, s_2) is in **R**
  - every state s_2 in S_2 has a state s_1 in S_1 and (s_1, s_2) is in **R**

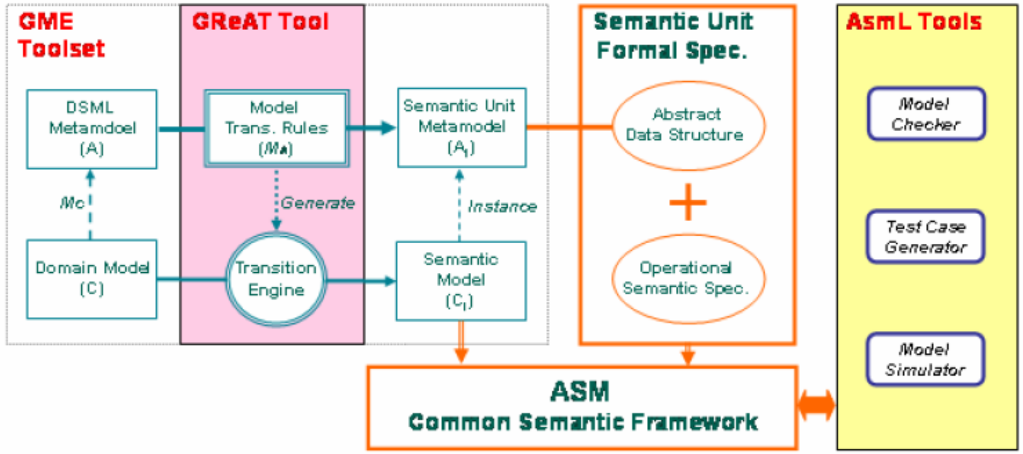
Checking Bisimilarity

- At the end of the transformation, the *cross-links* are preserved and sent to the bisimilarity checker, which performs the following steps
  - For every transition ***t*** :$S_{SC} \rightarrow S_{SC}'$ in the Statechart, find the equivalent transition ***t'*** :$S_{EHA} \rightarrow S_{EHA}'$ in the EHA
  - Check if S_{SC} and S_{EHA} are equivalent
  - Check if S_{SC}' and S_{EHA}' are equivalent
- The result of the bisimilarity checker will guarantee whether the results of the analysis on the analysis model are valid on the design model

Approach (2): Certification via semantic anchoring

- Problem description:
  - Statechart-X to Statechart-Y transformation
- Background: Semantic Anchoring
- Checking *weak* bisimilarity between semantically-anchored models

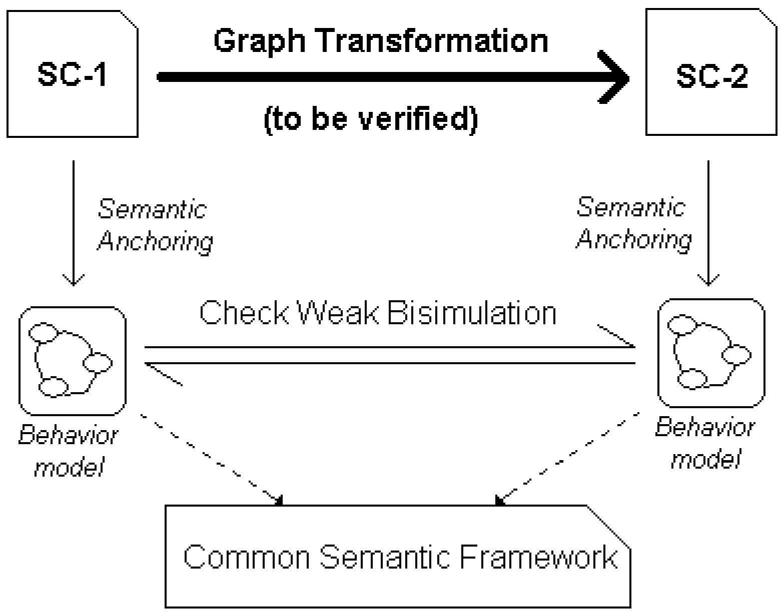
Background
Semantic Anchoring



- Semantic unit: well-defined, accepted 'unit' of semantics. E.g.: finite transition system
- Semantics of a DSML is *formally* defined by the **transformation** that maps <u>models</u> in the DSML into <u>configurations</u> of the semantic unit.

Specific Problem: Model-to-model transformation

- Both DSML-s (variants of Statecharts) are defined using semantic anchoring (i.e. via anchoring transformations *)
- They map to a common semantic framework ('semantic unit')
- **Concept:**
  1. Translate the source and target models using semantic anchoring to their behavior models
  2. Check for weak bisimilarity between the configured semantic units



*Kai Chen, Janos Sztipanovits, Sherif Abdelwahed, and Ethan K. Jackson. Semantic anchoring with model transformations. In ECMDA-FA, pages 115–129, 2005.

Bisimilarity

Bisimulation [San04] is defined for Labeled Transitions Systems (LTS). Given an LTS (S, Λ, →), a relation R over S is a *bisimulation* if:

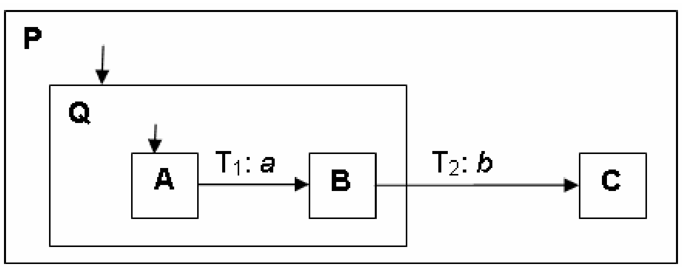
$$(p, q) \in R \text{ and } p \xrightarrow{\alpha} p' \text{ implies that there exists a } q' \in S \text{ such that } q \xrightarrow{\alpha} q' \text{ and } (p', q') \in R,$$

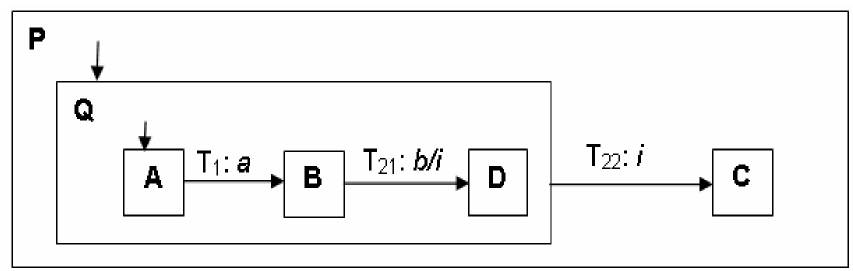
and conversely,

$$q \xrightarrow{\alpha} q' \text{ implies that there exists a } p' \in S \text{ such that } p \xrightarrow{\alpha} p' \text{ and } (p', q') \in R.$$

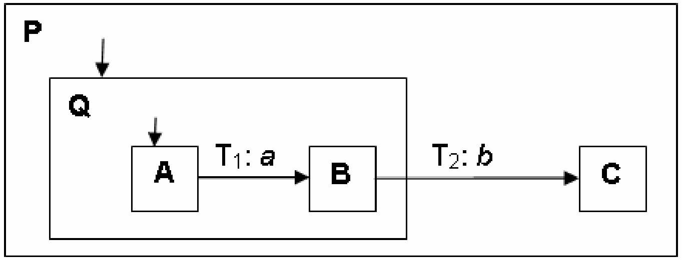
Example:

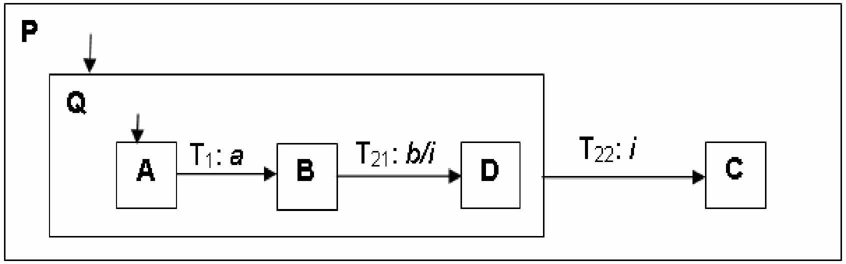
Statechart variants with (V1) and without (V2) inter-level transitions





The problem of behavioral bisimilarity





- For proper translation in V2 we need ‘instantaneous’ states (D) and actions (i)
  - I-state: can be entered and exited in the same step. A step is not complete until there are no I-states in the state configuration.
  - I-action: action executed (event posted and event triggers a transition) in the same step.
- (T_{21},T_{22}): macro-step:
  - D and i are invisible to the external observer
  - Executed as one, indivisible step

The semantic unit: FSM

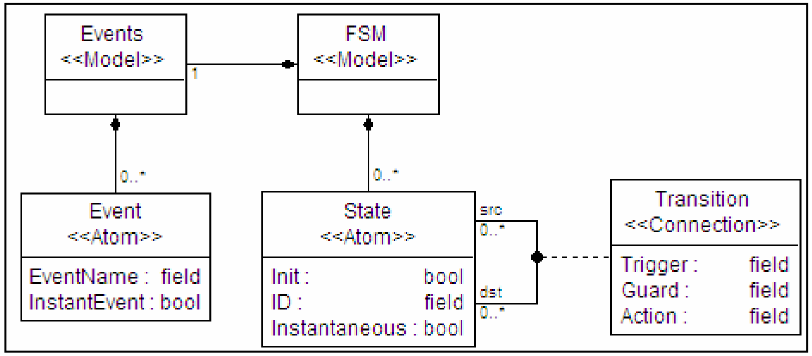
- Implemented in ASML
  - Executable specification language based on the Abstract State Machine concepts of Gurevich
- The S/A transformation ‘instantiates’ the semantic unit

```
interface Event
structure ModelEvent implements Event
structure LocalEvent implements Event
structure InstantEvent implements Event

class FSM
id as String
var outputEvents as Seq of ModelEvent
var localEvents as Set of LocalEvent
...

class State
id as String
var active as Boolean = false
var instantaneous as Boolean
var outTransitions as Set of Transition
...
class Transition
...
```

Metamodel fragment for FSM:



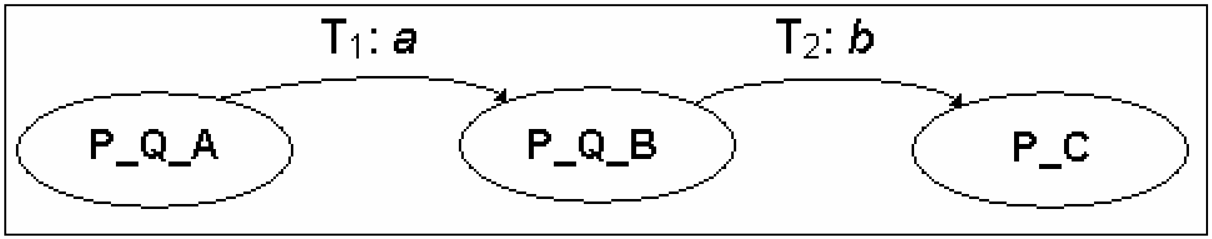
Setting up the V1/V2 transformation

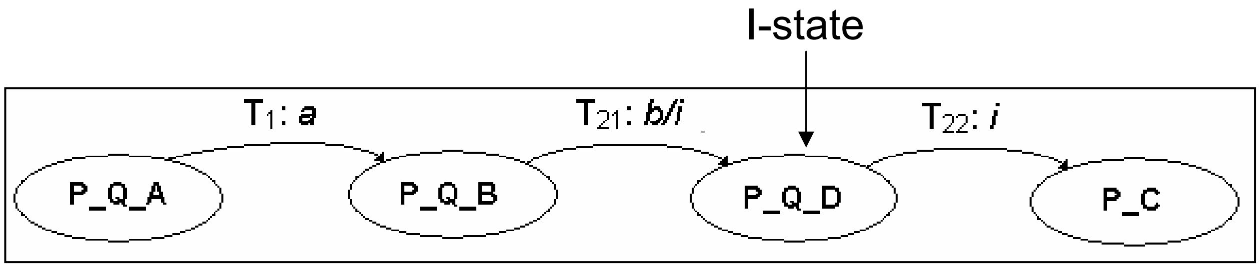
Implemented in GReAT

- Copy each state from V1 into V2
  - Link the source and target states
- For each transition in V1 do:
  - If *src* and *dst* have the same parent state, copy
  - else
    - repeat
      - add a self-start (or self-termination) state to the deeper of the two states, and
      - mark the parent as the source (or target)
    - until the source and target states are under the same parent

Verifying behavior preservation
Weak bisimilarity

Source and target FSMs:





Case Study: Behavior preservation

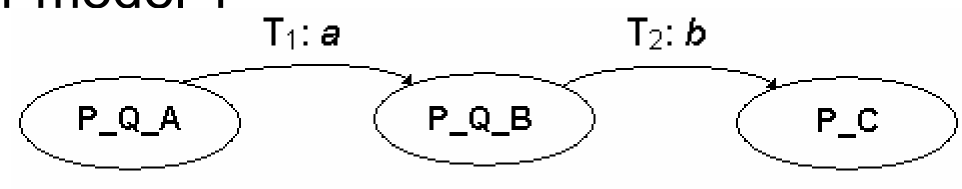
- Define Weak Bisimulation
  - Use the encoded labels of the FSMs to define the relation R
  - For all states (p, q) in R, and for all α: p $\stackrel{\alpha}{\Rightarrow}$ p', there exists a q' such that q $\stackrel{\alpha}{\Rightarrow}$ q' and (p', q') is in R
  - And conversely, for all α: q $\stackrel{\alpha}{\Rightarrow}$ q', there exists a p' such that p $\stackrel{\alpha}{\Rightarrow}$ p' and (p', q') is in R
  - p, q, p', q' are all non-instantaneous states (we *ignore* instantaneous states)
  - $\Rightarrow$ is a series of transitions between non-instantaneous states
  - α is the collection of actions and triggers in $\Rightarrow$, ignoring all instantaneous events

Checking for weak bisimilarity

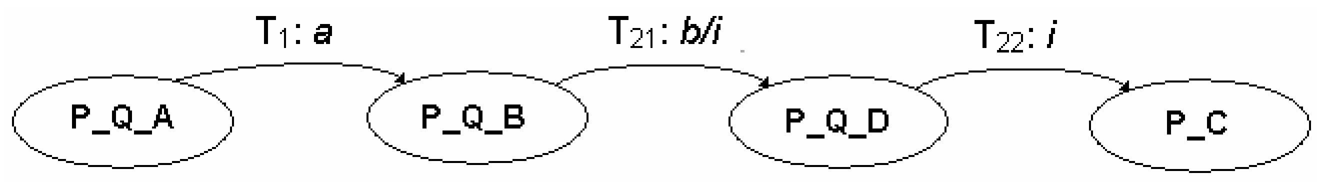
- Reduce the FSM to non-I-states and I–transitions:
  - Aggregate all sequences of transitions through I-states
- Establish R:
  - p (V1) and q (V2) are in R if they have the same label
  - During transformation labels are created s.t. labels in V2 are derived from labels in V1. The S/A GT uses a similar technique to generate labels for FSM states.
  - List all states with their transitions in a table, check that the weak bisimilarity relation holds for each state pair.

Case Study: Behavior Preservation

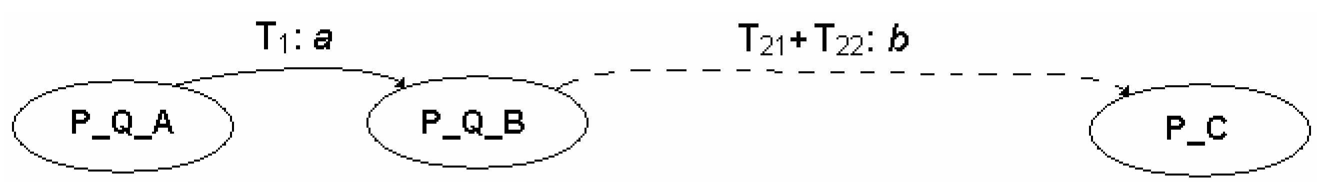
- Behavior model 1



- Behavior model 2



- Behavior model 2 with weak transition
  - Ignore instantaneous state P_Q_D and instantaneous action i

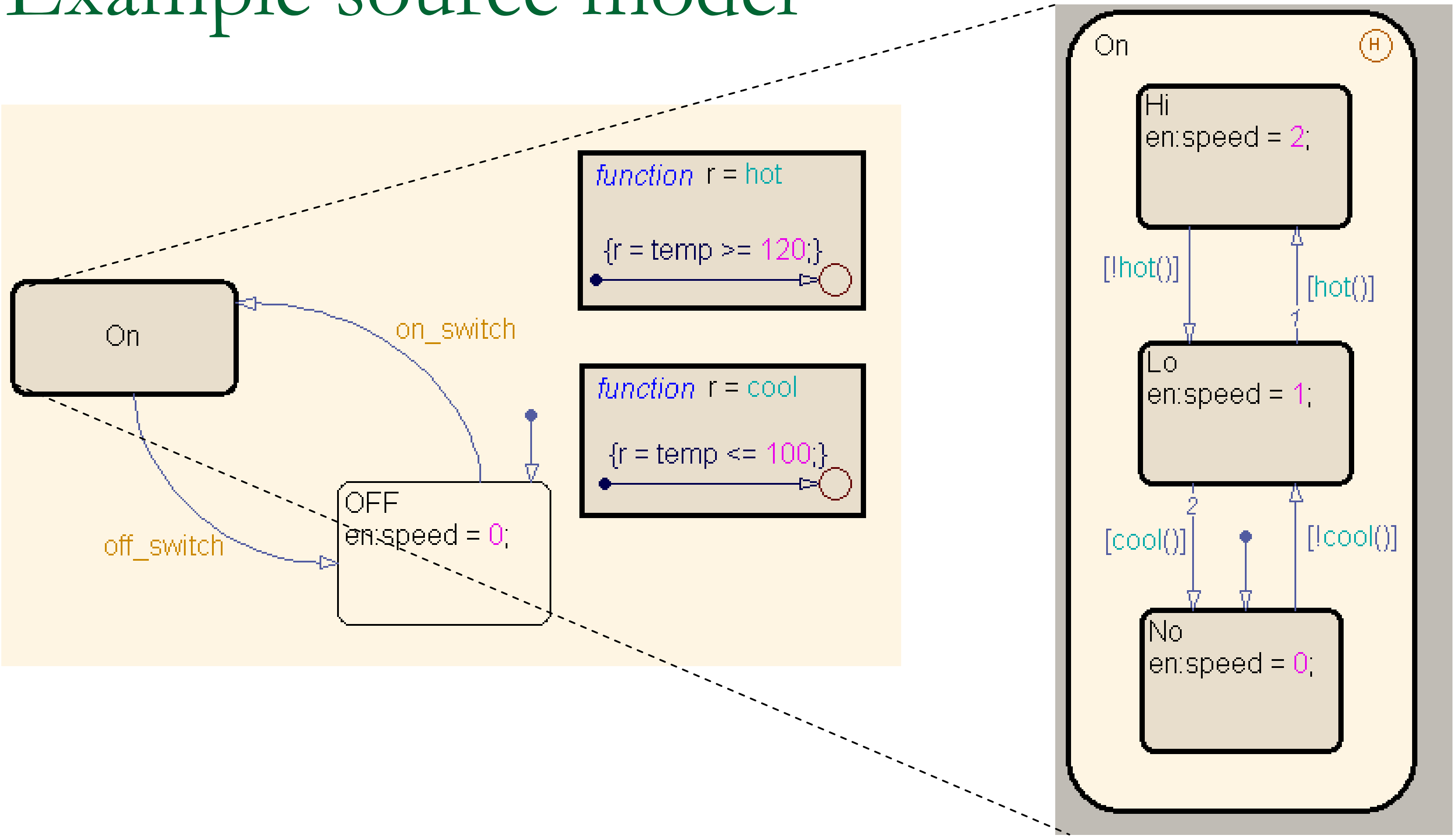


- The two systems are weakly bisimilar

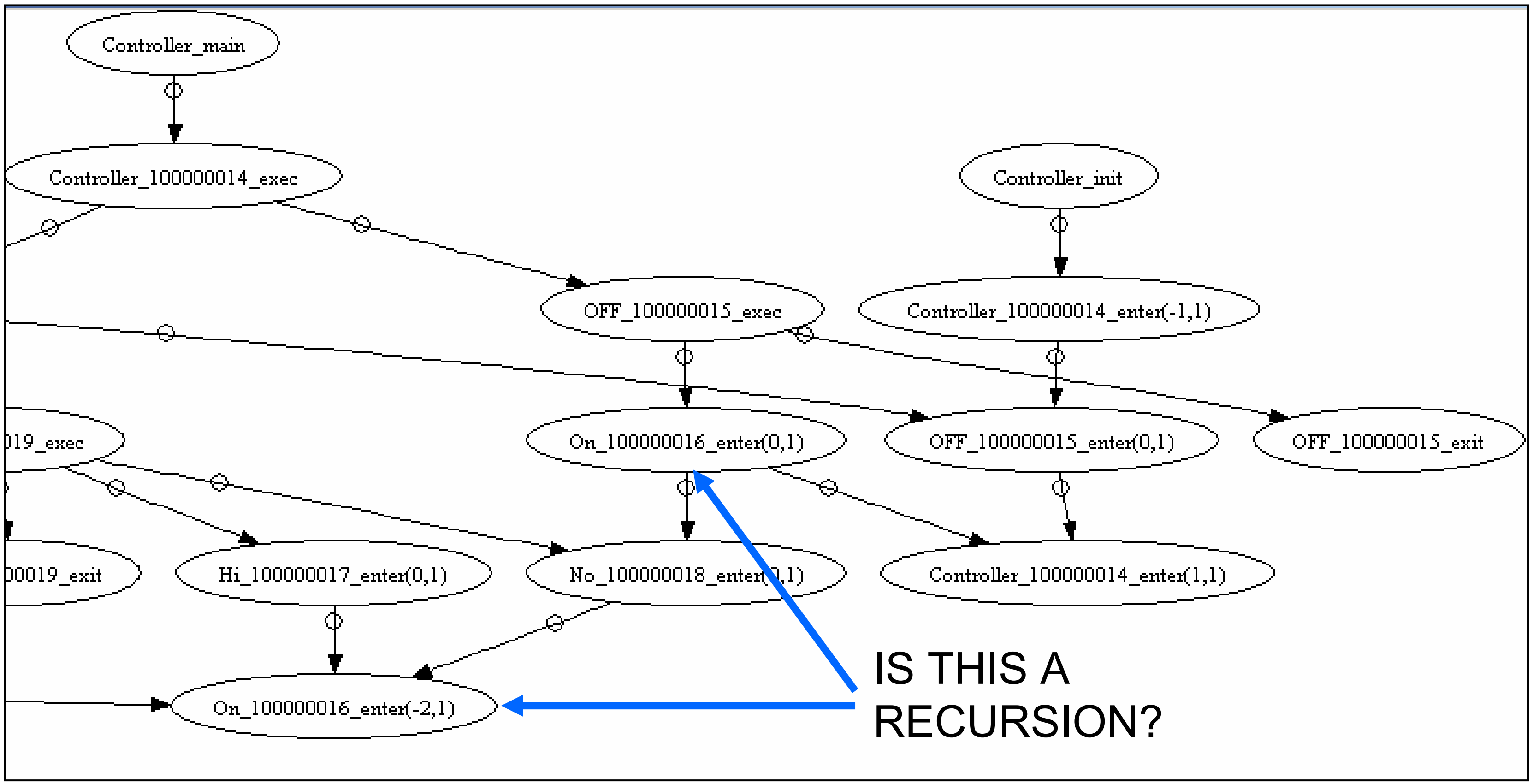
Exercise Problem

- Tool:
  - Stateflow -> C code generator
- Objective:
  - Show that the generated code uses a bounded amount of stack space (no infinite recursion)
- Problem:
  - Stateflow semantics proscribes enter/exec/exit actions on each state (including hierarchical ones)

Exercise Problem: Example source model



Exercise Problem:
Call graph from generated code



Exercise Problem: Thoughts

- It is *not* a recursion because the same routine entered in a different 'state' of the code/system
  - Different parameter values
  - Different state variable values
- How to verify the claim?
  - Model checking?
  - Theorem proving?

Summary

- Correctness of MT-s is essential for model-based development of embedded systems
- Instance-based verification is a pragmatic approach that also provides arguments for certifying the generated code
- Generating bisimilarity-based certificates help showing the reachability-oriented behavioral equivalence between different variants of Statecharts
- Many open research questions remain:
  - Extension to other models (e.g. timed automata, P/N)
  - Generalization to other kinds of properties
  - Other modeling languages, semantic units, verification tools