Tests and Proofs for Code Generators

Prahlad Sampath

Joint work with

A C Rajeev, K C Shashidhar (MPI), S Ramesh

General Motors

Code Generator

- Code generators are tools that take as input "models" in a modelling language and output various artifacts:
  - Code
  - Other models (one man's model is another man's code)

- Examples of code-generators
  - Rhapsody code-generator
  - Matlab/Stateflow simulator
  - Lex/Yacc
  - Query optimizers
  - ...

Why Code Generators

- More and more complex systems
- Difficult to program, and error-prone
- Avoid reinventing the wheel
  - Design-patterns that are well understood
  - Solutions that can be tuned
- Split up the verification process
  - Verify models, where domain specific abstractions can effectively simplify verification process.
  - Verify the code-generator – low level implementation details need to be considered only in this step
- Code-generators are the new compilers!

Verification of Code-Generators

- Approaches for verifying generated software:
  - Just ship the product!
  - Test the code
  - Model based testing
    - Test the equivalence of the “golden” model and the generated code
  - Equivalence checking for each run
    - Prove the equivalence of the model and the generated code
  - Testing using Automatic Test-case Generation
    - Generate test-cases for the code-generator
  - Formal verification
    - Prove the correctness of the code-generator
    - Does not scale to industrial tools (yet!)

The Tradeoffs

- White-box vs. Black-box
- One-off vs. Each-run
- Certification vs. Proof
- Other issues
  - Push-button vs. interactive
  - Portable / Reusable artifacts (eg. Test suites)
  - Tuneable

The difference

- Proving a code generator

  $\forall$ m:models, $\forall$ i:inputs:

  ModelExec(m, i) $\approx$ CodeExec(CodeGen(m),i)

- Testing a code generator

  Formany m:models, Formany i:inputs:

  ModelExec(m, i) $\approx$ CodeExec(CodeGen(m),i)

- Translation validation : fix a model m

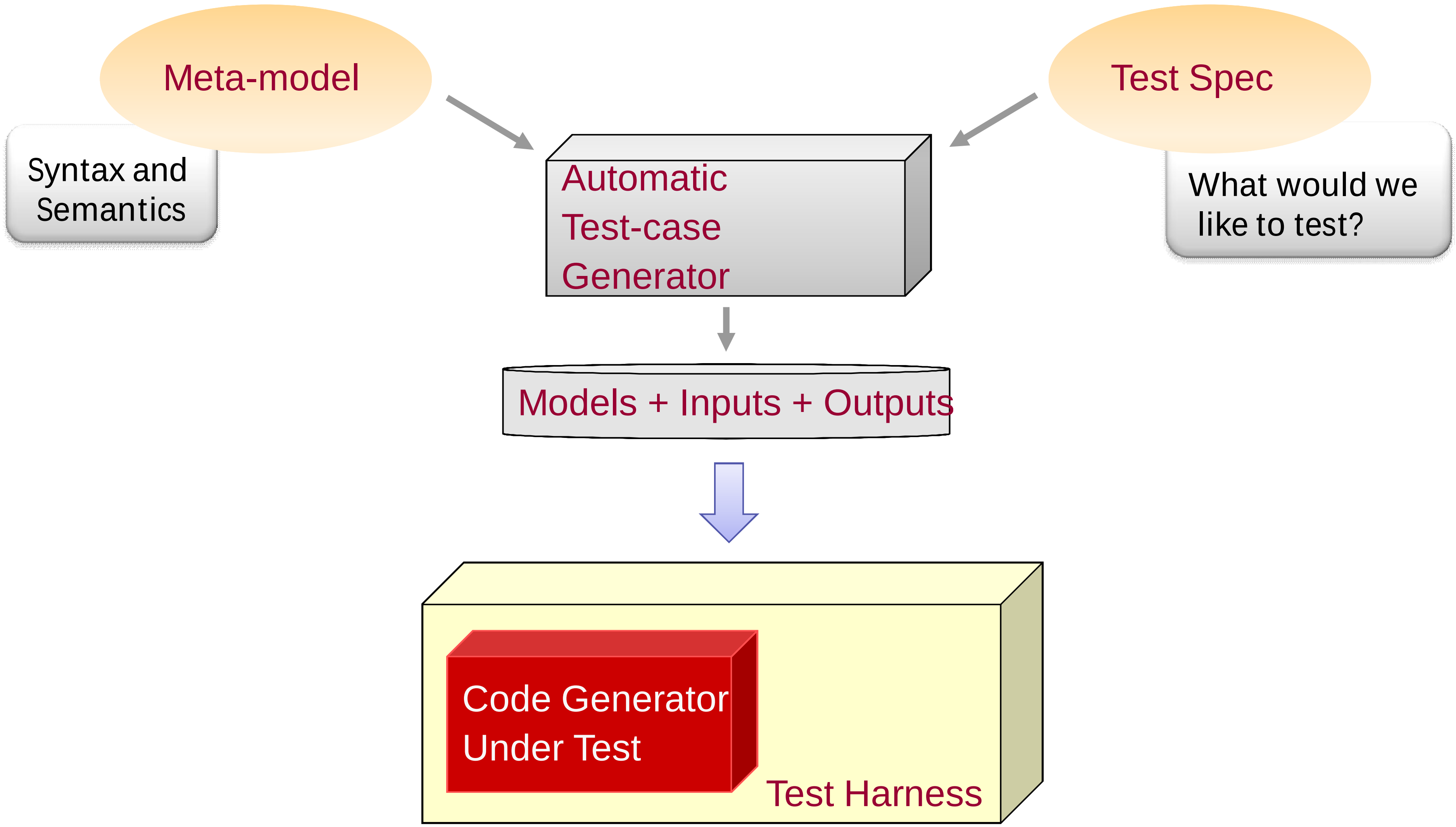
  $\forall$ i:inputs: ModelExec(m, i) $\approx$ CodeExec(CodeGen(m),i)

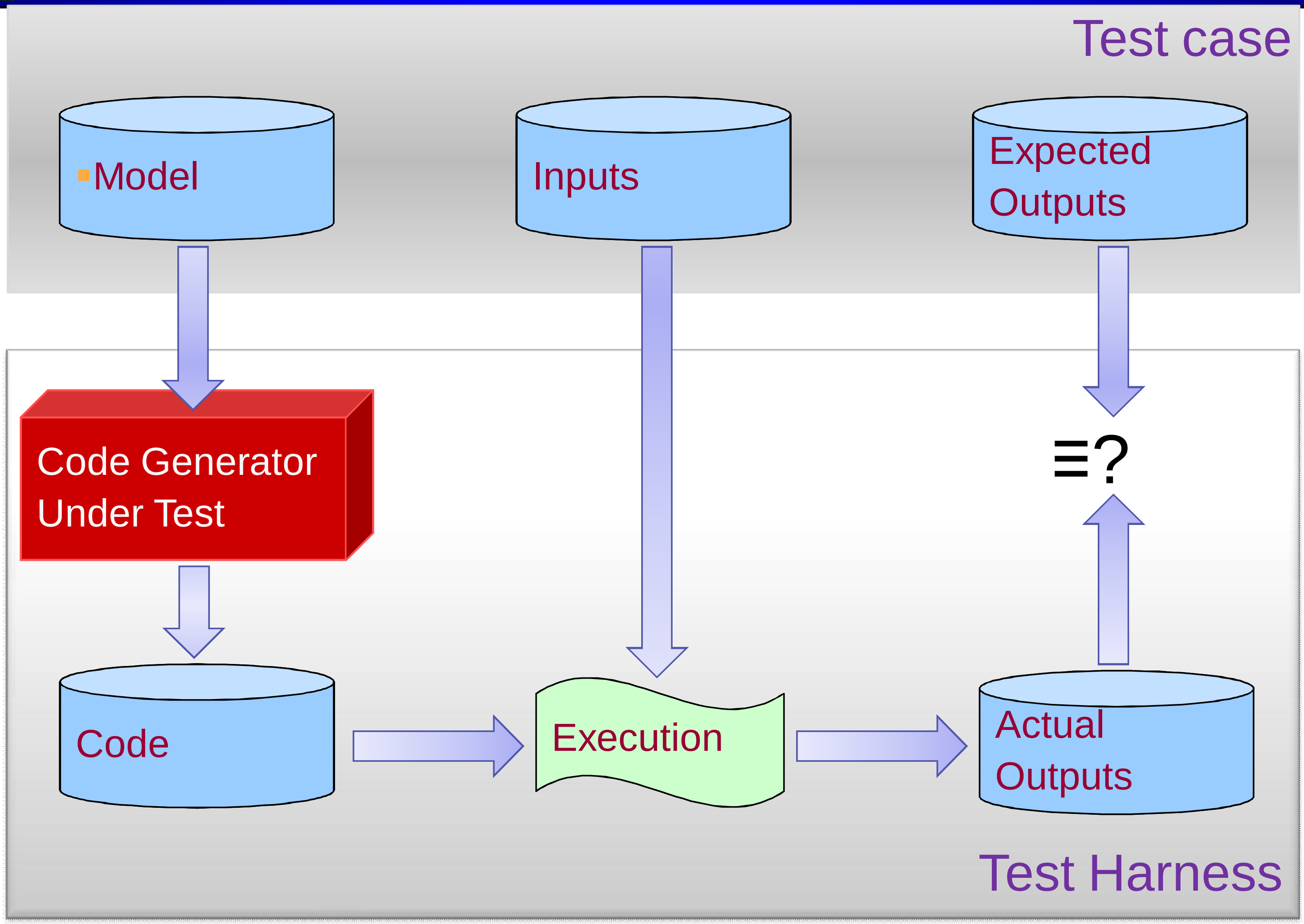
Testing Code Generators

- Syntax based
  - Generate models that cover syntactic elements
  - Generate Code
  - Perform model-based testing
    - Generate a large test-suite that achieves various coverage criteria over the model elements
- Issues
  - Very large test-suites.
  - Difficult to avoid “duplicates”
  - Models that are syntactically very different may be very similar semantically.

Testing Code Generators

- Semantics based
  - Is the semantics of the generated code the same as the semantics of the model?
  - Identify coverage over semantics
  - For each semantic behaviour
    - Generate model+Input that will exhibit this behaviour
    - Generate expected output for this model+input


Meta-model
Syntax and Semantics
Test Spec
What would we like to test?
Automatic Test-case Generator
Models + Inputs + Outputs
Code Generator Under Test
Test Harness


Test case
Model
Inputs
Expected Outputs
Code Generator Under Test
≡?
Code
Execution
Actual Outputs
Test Harness

Coverage of Semantics

- Represent semantics as “inference-rules”
- Achieve coverage of rules in “proofs”

- Proof-rules for Hoare logic

Coverage of Semantics

- Generate inference trees from rules that achieve coverage of rules

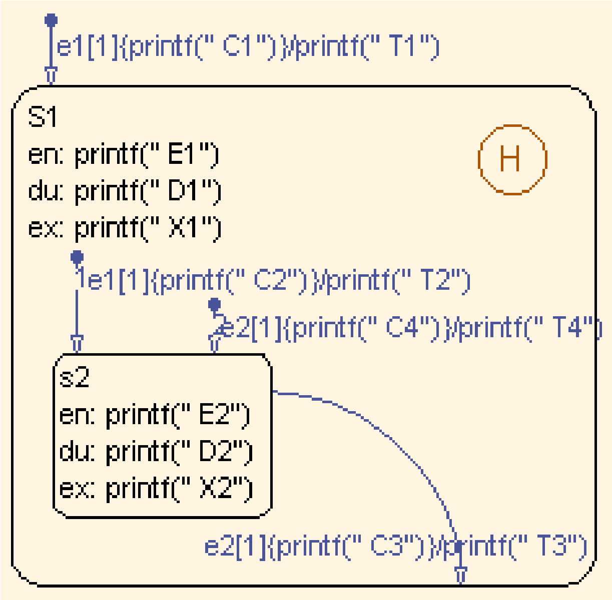
- Generate model+input that would give exhibit behaviour in inference tree
  - Reverse semantics!

- Generate expected output for given model+input

- Bundle the three into a test-case

Stateflow Semantics

- Inference rules for Stateflow:



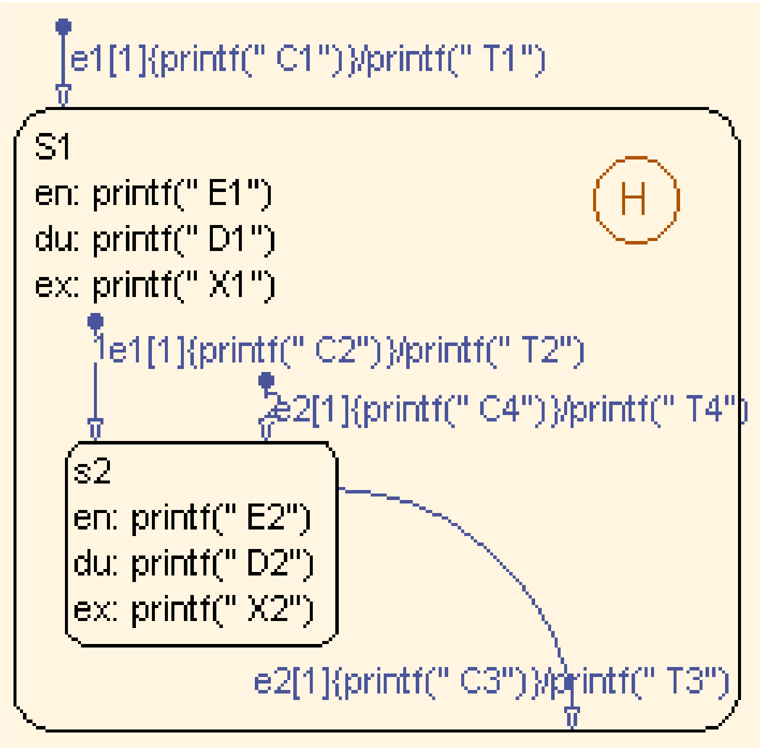
- Entering an atomic state s by a transition

$$\frac{\{P\}\ \mathsf{entryAct}(s)\ \{Q \rhd \Psi'\}}{(\!|\Psi \lhd P|\!)\ \tau \Rightarrow\!\Box\ s\ (\!|Q \rhd \Psi'|\!)}\ (\text{Atom-E})$$

- Entering an OR state by a transition, and its child state by default transition

$$\frac{\{P\}\ \mathsf{entryAct}(s)\ \{P_0 \rhd \Psi_0\} \qquad (\!|\Psi \lhd P_0|\!)\ \models_s \mathcal{T}_d\ (\!|P_1 \rhd \Psi_1|\!) \qquad (\!|\Psi \lhd P_1|\!)\ \Rightarrow\!\boxdot\ s_1\ (\!|Q \rhd \Psi_2|\!)}{(\!|\Psi \lhd P|\!)\ \tau \Rightarrow\!\Box\ s\ (\!|Q \rhd \cup_{k=0}^{2}\Psi_k|\!)}\ (\text{OR-dE-E})$$

Test-cases



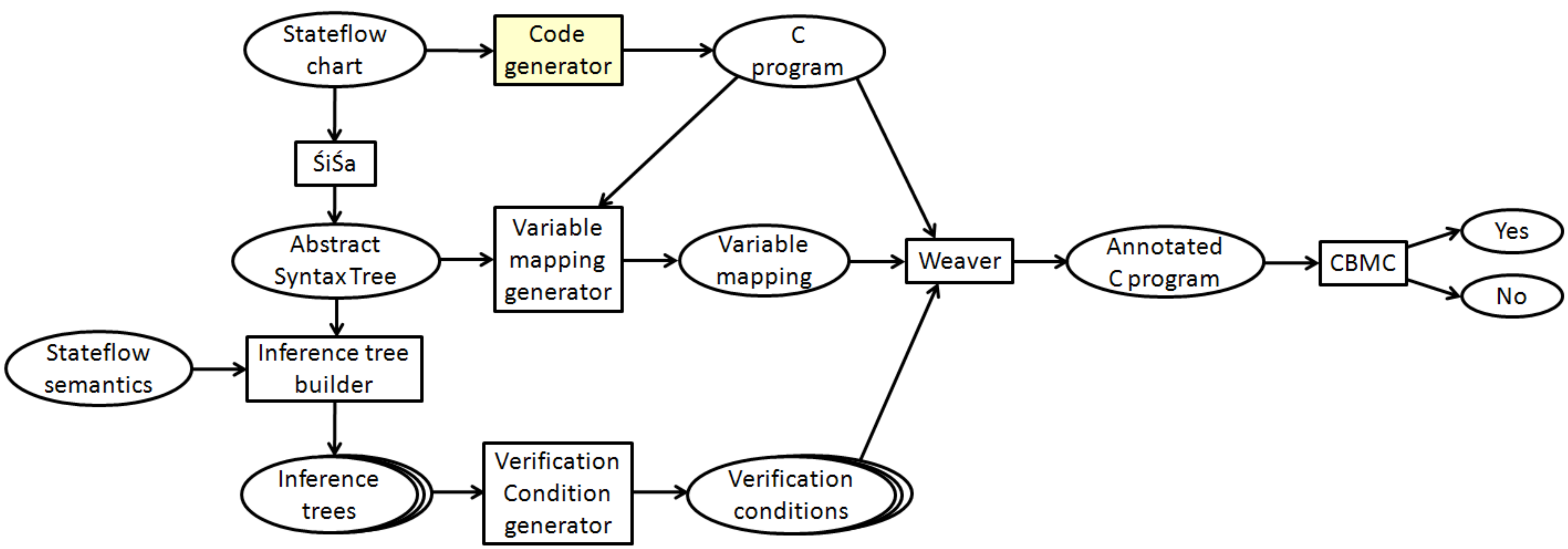
- Input events: e1, e2
- Expected actions: $\langle C1\,T1\,E1\,C2\,T2\,E2\,D1\,C3\,X2\,T3\,E2\rangle$

V6.2.1: $\langle C1\,T1\,E1\,C2\,T2\,E2\,D1\,C3\,X2\,T3\,C4\,T4\,E2\rangle$

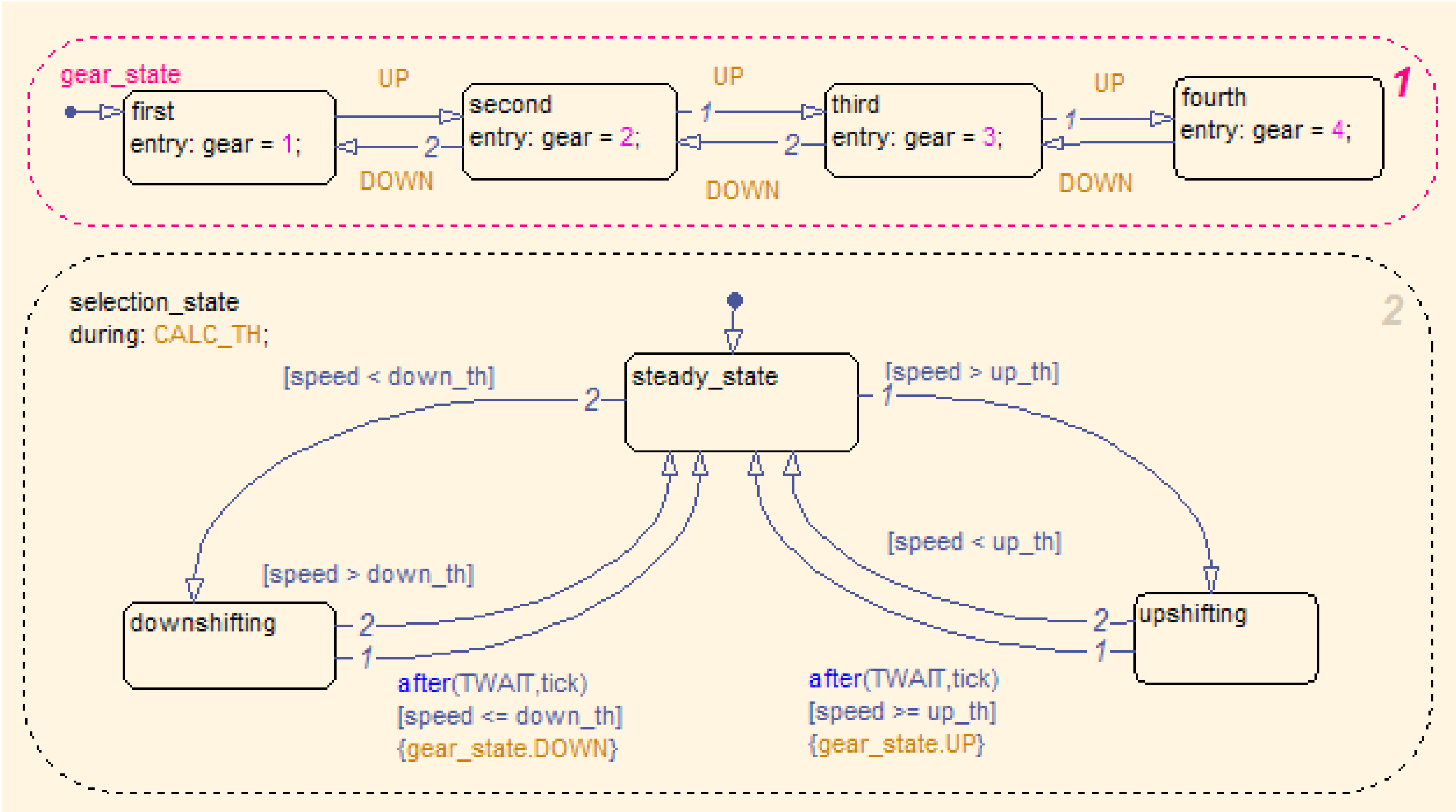
Proofs for Code Generators

- Translation validation approach
  - Calculate the semantics of the model as a set of inference-trees
  - Generate a verification condition (pre/post pair) from each inference tree
  - Verify these (pre/post) pairs on the program
  - Push-button on every codegen run
- Issues
  - What happens if translation-validation fails?
  - Assumes semantics is finite
  - Assumes that semantics is known!
  - Program verification can prove pre/post pairs
  - Assumes a business need for all this extra effort
  - Simulink/Stateflow – a sweet-spot

Translation Validation of Stateflow



Example



- Around 40 verification conditions
- Verified on generated c-code using cbmc